DATA REPRESENTATION FOR EFFICIENT AND RELIABLE STORAGE IN FLASH MEMORIES

A Dissertation

by

YUE WANG

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Anxiao (Andrew) Jiang
Committee Members,	Andreas Klappenecker
	Jennifer Welch
	Henry Pfister
Head of Department,	Hank Walker

May 2013

Major Subject: Computer Science

ABSTRACT

Recent years have witnessed a proliferation of flash memories as an emerging storage technology with wide applications in many important areas. Like magnetic recording and optimal recording, flash memories have their own distinct properties and usage environment, which introduce very interesting new challenges for data storage. They include accurate programming without overshooting, error correction, reliable writing data to flash memories under low-voltages and file recovery for flash memories. Solutions to these problems can significantly improve the longevity and performance of the storage systems based on flash memories.

In this work, we explore several new data representation techniques for efficient and reliable data storage in flash memories. First, we present a new data representation scheme—*rank modulation with multiplicity*—to eliminate the overshooting and charge leakage problems for flash memories. Next, we study the *Half-Wits*—stochastic behavior of writing data to embedded flash memories at voltages lower than recommended by a microcontroller's specifications—and propose three software-only algorithms that enable reliable storage at low voltages without modifying hardware, which can reduce energy consumption by 30%. Then, we address the file erasures recovery problem in flash memories. Instead of only using traditional error-correcting codes, we design a new *content-assisted decoder* (CAD) to recover text files. The new CAD can be combined with the existing error-correcting codes and the experiment results show CAD outperforms the traditional error-correcting codes.

ACKNOWLEDGEMENTS

Foremost, the greatest gratitude is extended to my advisor, Dr. Anxiao (Andrew) Jiang, for his thoughtful advice and guidance. He quickly became for me the role model of a successful researcher in the field. His dedication and passion on research and education influenced me positively. His insights and perception on novel approaches as well as on issues and challenges of active research areas inspired me tremendously. He is open-minded and caring for students, which helps make my research experience focused and fruitful. It is a great honor and pleasure to work with him. Without Andrew's encouragement and help, this thesis would not have been possible.

I would like to express my appreciation to Dr. Andreas Klappenecker, Dr. Jennifer Welch and Dr. Henry Pfister for serving on my degree committee. Their advice, feedback, and encouragement have been invaluable.

In addition, I am indebted to my peer colleagues, Fenghui Zhang, Hao Li, Vishal Kapoor, Shoeb Ahmed Mohammed, Qing Li and Yue Li for providing a stimulating and fun environment in which to learn and grow. Those great discussions and fun learning times will be memorable for years to come. I am especially thankful to Hao Li who is always willing to share his experiences and provide help not only in research but also in campus life.

Lastly, and most importantly, I must thank my parents and my husband, Yu Zhu, for their unflagging love and concern. Without their support and encouragement, this dissertation was simply impossible and I could not have gone this far. I also greatly thank my son, Ryan, whose lovely and adorable smile ignites the light of my life. To them I dedicate my dissertation.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

The representation of data plays a key role in storage systems. The objective of this thesis focuses on data representation techniques for efficient and reliable storage in flash memories. In this chapter, we first introduce flash memories and their key properties, then point out the main challenges of flash memories. We also describe the contribution of our work to solve those challenges. In addition, we present an overview of related works on flash memories.

1.1 Flash Memories and Their Properties

Flash memory, invented by Dr. Fujio Masuoka, is a type of non-volatile memory that can be electrically erased and reprogrammed. Flash memory is considered by system designers as an almost "ideal" non-volatile memory because it can be electrically erased and programmed in-system, offer at the same time very high-density and low cost-per-bit, random access, bit alterability, short read/write times and cycle times, excellent reliability [9]. Flash memory is a milestone in the development of the data storage technology. Due to its high performance, the applications of flash memories have expanded widely in recent years, such as cell phones, portable media players, digital cameras, and in the latest netbooks, tablets, and e-book readers, it is also being utilized widely by the video gaming device industry, which make it become the dominating member in the family of non-volatile memories [26]. It is expected that the world wide flash memory market will reach $51.2 billion by 2015, and then constitute a 12 percent share of the total semiconductor market.

1.1.1 Flash Cell Structure

The basic storage unit in a flash memory is a floating-gate transistor [9]. We also call it a cell. Each memory cell resembles a standard MOSFET, except the transistor has two gates instead of one. On top is the control gate (CG), as in other MOS transistors, but below this there is a floating gate (FG) insulated all around by an oxide layer. The FG is interposed between the CG and the MOSFET channel. Because the FG is electrically isolated by its insulating layer, any electrons placed on it are trapped there and, under normal conditions, will not discharge for many years [42]. Figure 1.1 shows the structure of a cell. The threshold voltage of the memory cell can be altered by changing the amount of charge present between the gate and the channel. If no electron is on the floating gate, the threshold voltage is low, and the transistor is "on" under reading voltage, whereas with injecting many electrons in the floating gate, the threshold becomes high, and then the transistor is "off".

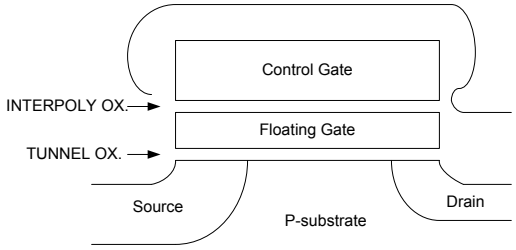


Figure 1.1: The structure of a flash cell.

The cell level is determined by the amount of charge trapped in the floating gate. Charge can be injected into the cell using the hot-electron injection mechanism or the Fowler-Nordheim tunnelling mechanism. The charge can also be removed from the cell using the Fowler-Nordheim tunnelling mechanism. By checking whether the

transistor is "on" or "off" in a single level cell (SLC), it can represent one bit of information. In a multi-level cell (MLC), which stores more than one bit per cell, the amount of current flow through the transistor is sensed (rather than simply its presence or absence), in order to determine more precisely the level of charge on the floating gate. q-level cell can store $\log_2 q$ bits.

1.1.2 NOR and NAND Flash

There are two main types of flash memories: NOR flash and NAND flash.

In NOR gate flash, each cell has one end connected directly to ground, and the other end connected directly to a bit line. Therefore, it allows random access to its cells. NOR flash is commonly used in embedded applications requiring a discrete non-volatile memory device, such as mobile phones. NOR flash is faster, but it's also more expensive.

A NAND flash partitions every block into multiple sections called pages, and a page is the unit of a read or write operation. Compared to NOR flash, NAND flash has the advantage of higher cell density. However, it may be much more restrictive on how its pages can be programmed, such as allowing a page to be programmed only a few times before erasure [18]. NAND flash has found a market in devices to which large files are frequently uploaded and replaced, such as MP3 players, digital cameras and USB drives.

1.1.3 Basic Operations in Flash Memory

Flash memory has three basic operations.

1.1.3.1 Reading

The read operation is performed by applying to the cell a gate voltage that senses the current flowing through the device. In NOR type flash memory, each cell's level

can be read individually, where in NAND type flash memory, the cells connected in series must be read in series. The read operation is easy and fast in both types of flash memory.

1.1.3.2 Writing/Programming

When programming, charge is injected into the cell using the hot-electronic mechanism or Fowler-Nordheim tunneling mechanism by applying an appropriate voltage to the control gate. The cell of both NOR and NAND can be programmed individually and the process is easy and fast [9].

1.1.3.3 Erasing

A prominent property of flash memories is *block erasure*. Cells in a flash memory are organized into blocks, with each block containing 10^5 or so cells. The state of a cell can be raised individually (program operation). But to decrease a cell level, the flash memory needs to erase the whole block (i.e., lowering the states of all the cells to 0) and then re-program all the cells. Such an operation is called *block erasure*. A very large voltage of the opposite polarity is applied between the control gate, pulling the electrons off the floating gate through quantum tunneling to erase the whole block, which is slow and energy-intensive. The block erasure operation not only significantly reduces speed, but also reduces the lifetime of the flash memory. This is because a block can only endure about $10^4 \sim 10^6$ erasures, after which the block may break down. Since the breaking down of a single block can make the whole memroy stop working, it is important to balance the erasures performed to different blocks.

1.2 Challenges of Flash Memories

As mentioned in the previous section, block erasure is a fairly violent process. Every time the system erase a block, it slightly damages the insulating barriers. Usually, the lifetime of flash memory is 10^5 erasure cycles. Therefore, block erasures can substantially reduce the writing speed, reliability and longevity of flash memories. For storage schemes, it is important to minimize block erasures. Although flash memory has many advantages such as low cost per bit, high storage density, quick read and write operations over other non-volatile memories, its interesting feature of "block erasure" operation makes flash memory face some new challenges, which require new data representation and coding schemes for efficient and reliable storage in flash memory.

1.2.1 Accurate Programming without Overshooting

When programming a cell, the charge is injected into the cell, and the injected charge becomes trapped. The amount of charge in a cell determines its level. Fast and accurate programming schemes for multi-level flash memory are a topic of significant research and design efforts. The flash memory does not support charge removal from individual cells due to block erasure. Overshooting is very costly for programming because once the injected charge overshoots the target level, the block need to be erased and then reprogrammed. As a result, in the industry, to program a cell, a sequence of charge injection operations are used to shift the cell level cautiously and monotonically toward the target charge level from below, in order to avoid undesired global erasures in case of overshoots. Thus, the attempt to program a cell requires quite a few programming cycles.

It is interesting to study a new data representation scheme to avoid the problem of overshooting while programming cells. In this work, we will present a generalization

of rank modulation, called rank modulation with multiplicity, in which different cells can share the same rank. We focus on the rewriting of data based on this new scheme, and study its basic properties.

1.2.2 Asymmetric Errors in Flash Memories

Flash memory is a storage medium with asymmetric properties [26]. After cells are programmed, the data are not error-proof, because the cell levels can be changed by various errors over time. Some important error sources include *write disturb* and *read disturb* (disturbs caused by writing or reading), as well as leakage of charge from the cells (called *data retention* problem). The errors in the cell levels have an asymmetric distribution in the up and the down directions. Our research of rank modulation with multiplicity provide a solution to tolerate asymmetric errors better.

1.2.3 Reliable Storage for Low-Power Devices

While the reliability, low cost, and high storage density of flash memory make it a natural choice for embedded systems [27], its relatively high voltage requirement introduces challenges for energy-efficient designs aiming to maximize the system's effective lifetime(e.g., the run time on a typical battery whose voltage declines over time). Lowering the common supply voltage would allow the CPU to operate in a more energy efficient manner, but writes to the flash memory then become unreliable.

How to address the voltage limitations of flash memory and guarantee reliable flash writes under lower voltage is a prominent topic. In this thesis, we present software-only coding schemes to enable reliable storage at low voltages without modifying hardware. It includes three algorithms: *in-place writes*, *multiple-place writes*, and *RS-Berger codes*.

1.2.4 File Recovery for Non-Volatile Memories

Non-volatile memories, especially flash memories have emerged as a crucial technology for storage systems due to their excellent speed and storage capacity. However, accompanying the improvement in data density, the reliability issue of non-volatile memories are attracting more and more attention [23]. File recovery will be one of the biggest challenges for storage systems. The amount of stored data is increasing at an explosive rate, but the data are not constantly checked to verify their reliability. The needed bit-error rate after decoding is 10^{-20} for storage systems. However, with the existing flash memories technologies, it cannot be achieved unless extra long error-correcting codes with substantial redundancy are used, which is impractical. However, it is not difficult for storage systems to achieve much higher bit-error rates, such as 10^{-3}.

We are interested in designing a content-based file recovery systems such that as long as the conventional error-correcting codes can reduced the bit-error/erasure rate to 10^{-3} after decoding, our file-recover system can practically recover the original files completely.

1.3 Contributions of This Work

In this thesis, we address the challenges facing flash memories by three techniques: A new data representation scheme for flash memories called *rank modulation with multiplicity* to eliminate overshooting and charge leakage problems; *Half-Wits*, a set of algorithms to enable reliable writes to flash memories while coping with low voltage; *Content-assisted file decoding* algorithms to make data storage in flash memories reliable. In the following, we introduce the three topics.

1.3.1 Rank Modulation with Multiplicity

Rank modulation, a new data representation scheme, is proposed to eliminate both the problem of overshooting while programming cells and the problem of memory endurance in aging devices [27]. This work proposes a generalization of rank modulation, called rank modulation with multiplicity, in which different cells can share the same rank.

We focus on the rewriting of data based on this new scheme. We study its basic properties, including the rewriting cost, optimal ways to change rank modulation states, and the expansion of rank modulation states given the rewriting cost. We consider two rewriting cost: unweighted and weighted rewriting cost and describe the analysis respectively. This work has been published in ACTEMT [30].

1.3.2 Half-Wits: Software Techniques for Embedded Flash Storage at Low Voltages

This work analyzes the stochastic behavior of writing to embedded flash memory at voltages lower than recommended by a microcontroller's specifications to reduce energy consumption. Flash memory integrated *within* a microcontroller typically requires the entire chip to operate on a common supply voltage almost double what the CPU portion requires. Our approach tolerates a lower supply voltage so that the CPU may operate in a more energy efficient manner. Energy efficient coding algorithms then cope with flash memory that behaves unpredictably.

The software-only coding algirhtms proposed in this work (*in-place writes*, *multiple-place writes*, *RS-Berger codes*) enable reliable storage at low voltages on unmodified hardware by exploiting the electrically cumulative nature of half-written data in write-once bits. For a sensor monitoring application using the MSP430, coding with in-place writes reduces the overall energy consumption by 34%. In-place writes are competitive when the time spent on low-voltage operations such as computation are

at least four times greater than the time spent on writes to flash memory. The evaluation of the proposed schemes shows that tightly maintaining the digital abstraction for storage in embedded flash memory comes at a significant cost to energy consumption with minimal gain in reliability. This work has been published in USENIX FAST [47].

1.3.3 Content-Assisted File Decoding for Non-Volatile Memories

To address the file recovery problem for data storage in non-volatile memories such as flash memories, we propose a *content-assisted decoding* (CAD) method for erasures recovery, which can be combined with existing storage solutions for text files. We preload the dictionaries that include the frequency information of words and phrases in the text of a given language. Thanks to the random and fast access features in flash memories, our proposed decoder gets the statistical information from the dictionaries quickly, then split the whole input noisy codeword into small pieces of subcodewords. Each subcodeword can be decoded into a word in the text, and the whole noisy codeword is recovered to form a most likely word sequences.

The CAD is modelled as a solution to an optimization problem, which mainly includes two parts: (1) segment the whole noisy codeword into a sequence of subcodewords and each subcodeword has a set of candidate words to decode; (2) choose the most likely word in the candidate word set for each subcodeword to form the most likely word sequence, which is a recovery for the original text file. Each part is also defined as an optimization problem and the dynamic programming algorithms are designed to get the solutions. The evaluation of the proposed methods with a set of benchmark files shows CAD can provide better erasure recovery capacity than the traditional ECC. This work has been published in [34].

1.4 Related Works

With the increasing importance of flash memories, numerous research work and accomplishments in flash memories have been published. The work in this thesis relates to a number of important research areas. They include rank modulation for flash memories, storage for low-power embedded devices, as well as error-correcting codes for flash memories and embedded systems.

1.4.1 Rank Modulation for Flash Memories

Rank modulation is a scheme that uses the relative order of cells, instead their absolute values, to represent data. It is first proposed and studies in [27, 28]. In addition to rewriting [27] and error correction [28], a family of Gray codes for rank modulation are also presented in [27]. A drawback to the rank-modulation scheme is the need for a large number of comparisons when reading the induced permutation from a set of n cell-charge levels. Instead, in a recent work [57], the n cells are locally viewed through a sliding window resulting in a sequence of small permutations which require less comparisons. Based on [57], gray codes are studied for the local rank modulation scheme in order to simulate conventional multi-level flash cells while retaining the benefits of rank modulation in [49, 15, 16].

The encoding approach for rank modulation in [27, 28] is based on the "push to the top" operation, which raises the charge level of a single cell above the rest of the cells. It is a good scheme that speeds up cell programming by eliminating the over shooting problems. However, it is not optimal in terms of minimizing the increase of cell levels. Gad presents a "minimal-push-up" operation model and proposes a compressed encoding for rank modulation in [17].

An extension for rank modulation is to use permutations of a given multiset to represent data. A series of papers [32, 31] discuss the permutation array under

the Chebysheve distance. Decoding algorithms for permutation arrays are proposed in [55, 51, 33] and the capacity of a new WAM code based on permutation arrays is studied in [14].

1.4.2 Storage for Low-Power Embedded Devices

Recent research focuses on optimizing use of off-chip flash memory. Off-chip memory allows for special features and larger memories than found on microcontrollers, but introduces additional costs for components. Microhash [58] is a memory index structure tailored for sensor devices with a large external flash memory. Mathur [39] perform an extensive study of available flash memory candidates for sensor devices and demonstrate that an off-chip parallel NAND flash memory decreases the energy consumption of storage. Considering the off-chip NAND flash memory as the best candidates for sensor devicse, Agrawal [3] proposes a method that allows sensor devices to exploit their flash memory while adapting to different amount of RAM. However, our storage schemes are designed for already deployed low-power devices that use on-chip flash memory. Moreover, while devices at the scale of sensor nodes might switch to block-grained, large off-chip flash memory, RFID-scale platforms might not benefit from this transition because of their challenging resource limitations to drive I/O.

1.4.3 Error-Correcting Codes for Storage

An error-correcting code is a system of adding redundant data, or parity data, to a message, such that it can be recovered by a receiver even when a number of errors (up to the capacity of the code being used) are introduced. Error-correcting codes are frequently used in reliable storage in media such as CDs, DVDs, hard disks and flash memories [35]. ECCs are usually categorized into convolutional codes and block codes. Convolutional codes work on bit or symbol streams of arbitrary length,

while block codes are processed on fixed-size blocks. Examples of block codes are repetition codes, Hamming codes, Reed-Solomon codes [44], turbo codes [7] and low-density parity-check codes (LDPC) [19, 36]. Those codes are widely used in binary symmetric channel (BSC) and not considering the asymmetric property.

With the rapid development of flash memories, there have been many research on error correction for flash memories. Most previously published flash error correction codes [11, 13, 22] are designed for NAND flash memory. Chen [12] mentions that NOR flash normally does not require error correction. The errors in flash cell levels often have an asymmetric property. These techniques consider neither the asymmetry in flash memory nor the resource limitations of low-power embedded devices. In [10], error-correcting codes that correct asymmetric errors of limited magnitude are designed and in [61], ECCs that correct different numbers of asymmetric errors depending on the codewords' Hamming weights are described for flash memories. Jiang [25] suggests error-correcting codes for multi-level cells (MLC) flash memory that cope well with the WOM property of flash memory and Zhou [60] discusses solutions by selecting dynamic reading thresholds to reduce the asymmetric errors due to voltage or resistance drift in flash memory.

Many previous codes leverage the fact that each cell of MLC flash memory represents more than one bit of information. But the fact that single-level cells (SLC) are more suitable for embedded devices, in addition to the occurrence of errors in low-voltage conditions, requires a reconsideration of these codes for SLCs at low voltage. Zemor [59] introduces error-correcting WOM codes for flash memory. They suggest codes that are able to correct up to one error when the flash memory is given enough voltage. This work does not account for errors that occur at low voltage. Godard [20] proposes hierarchical code correction and reliability management for NOR flash memory. This work considers on-chip ECCs such as Hamming codes to

correct the errors in NOR flash memory.

The rest of the thesis is organized as follows. Chapter 2 describes a new data representation scheme: rank modulation with multiplicity for flash memories. Chapter 3 focuses on software techniques for reliable embedded flash storage under low voltage. Chapter 4 presents content-assisted decoding algorithms for file recovery in non-volatile memories. The thesis closes in Chapter 5, where general concluding remarks and recommendations for future work are presented.

2. RANK MODULATION WITH MULTIPLICITY

In this chapter, we present a novel data representation scheme for multilevel flash memory cells—rank modulation with multiplicity—in which a set of n cells stores information according to their charge levels' relative order and multiple cells can share the same rank. We focus on the rewriting of data based on the new rank modulation scheme. We study its basic properties, including the rewriting cost, optimal ways to change rank-modulation states, and the expansion of rank modulation states given the rewriting cost.

2.1 Introduction

Flash memory is a dominant nonvolatile memory technology and a prominent candidate to replace the well-established magnetic recording technology in the near future due to its properties of high reliability and storage density, as well as relative low cost. A prominent property of flash memories is that although it is easy to increase a cell level, to decrease any cell level, a whole block of cells have to be erased and reprogrammed, which is very costly. Therefore, fast and accurate programming schemes for multilevel flash memories are a topic of significant research and design efforts. The programming cycle sequence is designed to cautiously approach the target charge level from below so as to avoid undesired global erases in case of overshoots. Consequently, these attempts still require many programming cycles, and they work only up to a moderate number of levels per cell. Besides of the need for accurate programming, another problem for multilevel flash cells is errors that originate from low memory endurance [9], by which a drift of threshold levels in aging devices may cause programming and read errors. To minimize the number of expensive block erasure operations caused by overshooting and to maintain the data

integrity, a new data representation scheme is needed for flash memories.

2.1.1 Rank Modulation for Flash Memories

Rank modulation is a scheme that uses the relative order of cell levels to represent data. Consider n cells $c_1, c_2, \ldots, c_n$ whose levels are $\ell_1, \ell_2, \ldots, \ell_n$, respectively, where $\ell_i \neq \ell_j$ when $i \neq j$. Let $(a_1, a_2, \ldots, a_n)$ be a permutation of the set $\{1, 2, \ldots, n\}$, such that $\ell_{a_1} > \ell_{a_2} > \cdots > \ell_{a_n}$. Then for $1 \leq i \leq n$, the cell c_{a_i} has the i-th highest level and is said to have *rank* i. The rank modulation scheme uses the ranks of cells (instead of the real values of the cell levels) to represent data; namely, the information bits are mapped to the permutation $(a_1, a_2, \ldots, a_n)$ [27]. In this way, no discrete cell levels are needed and only a basic charge-comparing operation is required to read the permutation. Rank modulation can make it simpler and more robust to program flash memory cells, where the cell levels are only allowed to monotonically increase during the programming process. Besides, it eliminates the overshooting problem in flash memory and reduces corruption due to retention.

2.1.2 Existing Codes for Rank Modulation

Rank modulation is first proposed in [27], in which balanced Gray codes are constructed. They also investigate rewriting schemes for random data modification and present both an optimal scheme for the worst case rewrite performance and an approximation scheme for the average-case rewrite performance [27].

Error-correcting codes are very important for rank modulation, and they have attracted interest among researchers. There have been some results on error-correcting codes for rank modulation equipped with the Kendall's τ-distance. In [29], an one-error-correcting code is constructed based on metric embedding, whose size is provably within half of the optimal size. In [5], the capacity of rank modulation codes is derived for the full range of minimum distance between codewords. There has also

been some work on error-correcting codes for rank modulation equipped with the L_∞ distance [54, 50]. The distance metric is more appropriate for cells where the noise in cell levels has limited magnitudes, called limited-magnitude rank-modulation codes. Some optimal codes for limited-magnitude errors are presented in [54, 50]. The systematic error-correcting codes for rank modulation is explored and proved to achieve the same capacity as general error-correcting codes in [62].

2.1.3 Rank Modulation's Drawback

Although rank modulation scheme is able to eliminate both the problem of overshooting while programming cells, and the problem of memory endurance in aging devices, it makes sacrifice of reducing the storage capacity. n cells with q levels can represent at most $\log_2 q^n$ information bits with the concrete cell levels representation schemes, however, using rank modulation scheme, it can only store at most $\log_2 n!$ bits.

2.1.4 Rank Modulation with Multiplicity

In order to improve the storage capacity of rank modulation, we study an extension of rank modulation, where multiple cells can have the same rank. The general idea is that we see cells of similar levels as having the same rank, and see cells of sufficiently different levels as having different ranks. There are naturally various ways to define the similarity of cell levels, including the following one. Let Δ and δ be two parameters, where $\Delta \geq \delta \geq 0$. For n cells whose levels can be ordered as $\ell_{a_1} \geq \ell_{a_2} \geq \cdots \geq \ell_{a_n}$, we require that for $1 \leq i < n$, either $\ell_{a_i} - \ell_{a_{i+1}} \leq \delta$ or $\ell_{a_i} - \ell_{a_{i+1}} > \Delta$. Then for $1 \leq i < n$, if $\ell_{a_i} - \ell_{a_{i+1}} \leq \delta$, we say the cells c_{a_i} and $c_{a_{i+1}}$ have the same *rank*; if $\ell_{a_i} - \ell_{a_{i+1}} > \Delta$, we say they have different *ranks*. For example, assume $\delta = 0.2$, $\Delta = 0.5$, $n = 8$ and $(\ell_1, \ldots, \ell_8) = (0.8,\ 2.2,\ 1.56,\ 0.21,\ 0.2,\ 2.1,\ 1.35,\ 1.38)$. Then $(a_1, \ldots, a_8) = (2, 6, 3, 8, 7, 1, 4, 5)$,

and $(\ell_{a_1}, \ldots, \ell_{a_8}) = (2.2,\ 2.1,\ 1.56,\ 1.38,\ 1.35,\ 0.8,\ 0.21,\ 0.2)$; so the cells c_2, c_6 have rank 1, c_3, c_8, c_7 have rank 2, c_1 has rank 3, and c_4, c_5 have rank 4. (We may further bound the maximum difference between the levels of the cells of the same rank.) Here the parameter Δ ensures the cell levels for different ranks are sufficiently apart so that they can tolerate noise better, and δ is chosen appropriately so that the cell levels for the same rank can be programmed successfully with high probability. Allowing cells to have the same rank can help achieve higher storage capacity. And since the gap between the cell levels of different ranks does not have a specific required value – in particular it is not upper bounded – the cells can still be programmed easily without the risk of charge overshooting (as long as the cell levels of each individual rank are programmed well.) We can use the same low-rank-to-high-rank method to program cells as in [27]. Note that when $\delta = \Delta = 0$, as no two cells can practically have exactly the same level, the scheme is reduced to the original rank modulation where every cell has a distinct rank [27].

Let

$$\begin{aligned}\mathcal{S}_n = &\{(s_1, s_2, \ldots, s_k) \mid 1 \leq k \leq n;\ s_i \subseteq \{1, 2, \ldots, n\} \text{ and } |s_i| \geq 1 \text{ for } 1 \leq i \leq k;\\ &\cup_{i=1}^{k} s_i = \{1, \ldots, n\};\ s_i \cap s_j = \emptyset \text{ for } i \neq j\}\end{aligned}$$

Every element $(s_1, s_2, \ldots, s_k)$ in $\mathcal{S}_n$ is a partition of the set $\{1, 2, \ldots, n\}$. We use $(s_1, s_2, \ldots, s_k)$ to denote the cells' ranks, where for $1 \leq i \leq k$, the cells with indices in s_i have the rank i. (For the previous example, we have $(s_1, s_2, \ldots, s_k) = (\{2, 6\}, \{3, 7, 8\}, \{1\}, \{4, 5\})$.) The data are represented by the elements of $\mathcal{S}_n$. Note that the difficulty of programming cells varies for the different elements of $\mathcal{S}_n$. It is simple to program two cells into different ranks since we only need the gap between their levels to be sufficiently large; but it is more challenging to program cells into

the same rank because their levels need to be similar. The more cells share the same rank, the more difficult it is to program them. In the following, we consider only the elements of $\mathcal{S}_n$ where every rank accommodates at most λ cells; that is, let

$$\mathcal{S}_{n,\lambda} = \{(s_1, s_2, \ldots, s_k) \in \mathcal{S}_n \mid \forall\ i,\ \ |s_i| \leq \lambda\}$$

and we use only the elements of $\mathcal{S}_{n,\lambda}$ to represent data. The parameter λ determines the tradeoff between the complexity of cell programming and the storage capacity. We call the scheme *rank modulation with multiplicity* λ.

The rank modulation with multiplicity λ uses the elements in $\mathcal{S}_{n,\lambda}$, called *rank-modulation states*, to represent data. Let $\mathcal{L} = \{0, 1, \ldots, L-1\}$ denote the alphabet of the stored data. Then there is a surjective map $\mathcal{D} : \mathcal{S}_{n,\lambda} \to \mathcal{L}$, such that the rank-modulation state $\mathbf{s} = (s_1, s_2, \ldots, s_k) \in \mathcal{S}_{n,\lambda}$ represents the data $\mathcal{D}(\mathbf{s}) \in \mathcal{L}$. The number of stored information bits, $\log_2 L$, can be maximized by letting $L = |\mathcal{S}_{n,\lambda}|$; and by letting $L < |\mathcal{S}_{n,\lambda}|$, the cost of rewriting data can be reduced.

Example1. *Let* $n = 3, \lambda = 2$. *Then* $\mathcal{S}_{n,\lambda} = \{(\{1\},\{2\},\{3\}), (\{1\},\{3\},\{2\}), (\{2\},\{1\},\{3\}), (\{2\},\{3\},\{1\}), (\{3\},\{1\},\{2\}), (\{3\},\{2\},\{1\}), (\{1\},\{2,3\}), (\{2\},\{1,3\}), (\{3\},\{1,2\}), (\{1,2\},\{3\}), (\{1,3\},\{2\}), (\{2,3\},\{1\})\}$. *So* $|\mathcal{S}_{3,2}| = 12$. *Up to* $\log_2 12$ *information bits can be stored.*

2.1.5 Storage Capacity Improvement by Rank Modulation with Multiplicity

The general value of $|\mathcal{S}_{n,\lambda}|$ can be computed by recursion:

$$|\mathcal{S}_{n,\lambda}| = \sum_{i=1}^{\min\{n,\lambda\}} \binom{n}{i} |\mathcal{S}_{n-i,\lambda}| \ \text{ for } n > 0; \text{ and } |\mathcal{S}_{0,\lambda}| = 1$$

We show $|\mathcal{S}_{n,\lambda}|$ for $2 \leq n \leq 16$ and $\lambda = 1, 2, 3, 4$ in Figure 2.1.

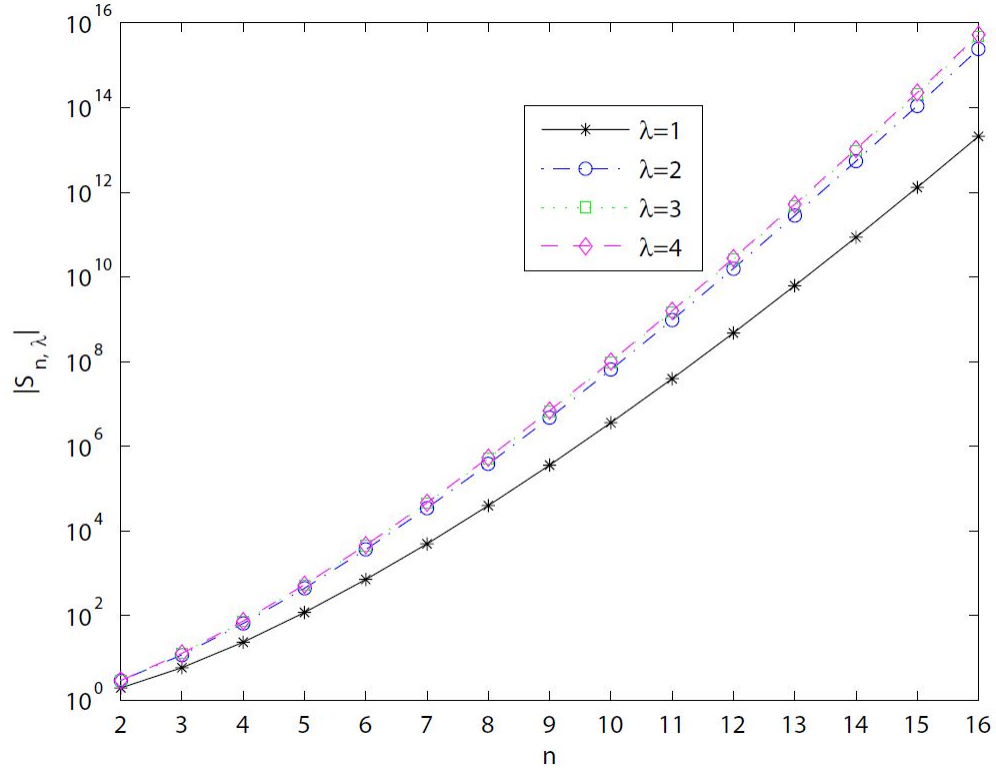


Figure 2.1: The value of $|\mathcal{S}_{n,\lambda}|$ for $\lambda = 1, 2, 3, 4$.

The plot for $\lambda = 1$ shows the maximum number of symbols it can represent by using the original rank modulation scheme. When $\lambda = 2, 3, 4$, the cardinality of $\mathcal{S}_{n,\lambda}$ is increased obviously shown in Figure 2.1.

2.2 Basic Operations

For the rewriting of data, we consider the memory model where the cell levels can only increase, not decrease. For flash memories, this is the way cells are programmed via charge injection (without the expensive block erasure operation). Let us define the basic operation we can use to change the rank-modulation state, in order to rewrite data. The basic operation is a "push operation", where we either push a cell to a higher rank (if there are fewer than λ cells of that rank), or push the cell to

the top so that it has a higher rank than all the other $n-1$ cells. More specifically, let $\mathbf{s} = (s_1, s_2, \ldots, s_k) \in \mathcal{S}_{n,\lambda}$ be a rank-modulation state. For any i, j such that $1 \leq i < j \leq k$ and $|s_i| < \lambda$, if $|s_j| > 1$, with a push operation, we can change $\mathbf{s}$ to

$$(s_1, \ldots, s_i \cup \{p\}, \ldots, s_j \setminus \{p\}, \ldots, s_k)$$

for some $p \in s_j$; if $|s_j| = 1$, we can change $\mathbf{s}$ to

$$(s_1, \ldots, s_i \cup \{p\}, \ldots, s_{j-1}, s_{j+1}, \ldots, s_k)$$

with p being the only element in s_j. And for any $i \in \{1, 2, \ldots, k\}$ such that $|s_i| > 1$, we can change $\mathbf{s}$ to

$$(\{p\}, s_1, \ldots, s_i \setminus \{p\}, \ldots, s_k)$$

for some $p \in s_i$. For any $i \in \{2, 3, \ldots, k\}$ such that $|s_i| = 1$, we can change $\mathbf{s}$ to

$$(\{p\}, s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_k)$$

with p being the only element in s_i. (Note that if $\lambda = 1$, the push operation here is reduced to the "push-to-top" operation for the original rank modulation scheme [27].)

2.3 Unweighted Rewriting Cost

For rewriting data, it is desirable to increase the cell levels as little as possible with each rewrite, so that more rewrites can be performed before the cell levels reach the maximum limit. (After that, the block erasure will be needed to lower the cell levels back to the minimum value.) So in this section, we consider the cost of changing the rank-modulation state from $\mathbf{s}$ to $\mathbf{s}'$ as the minimum number of push operations needed to change $\mathbf{s}$ to $\mathbf{s}'$, which we denote by $d(\mathbf{s}, \mathbf{s}')$. We call $d(\mathbf{s}, \mathbf{s}')$ the

unweighted rewriting cost. (A weighted version of the rewriting cost will be studied in the latter section.) It is not hard to see that

$$\max_{\mathbf{s},\mathbf{s}' \in \mathcal{S}_{n,\lambda}} d(\mathbf{s}, \mathbf{s}') = n - 1.$$

An example of $\mathbf{s}$ and $\mathbf{s}'$ that achieve this maximum unweighted rewriting cost, $d(\mathbf{s}, \mathbf{s}') = n-1$, is $\mathbf{s} = (\{1\}, \ldots, \{i-1\}, \{i\}, \{i+1\}, \ldots, \{n\})$ and $\mathbf{s}' = (\{1\}, \ldots, \{i-1\}, \{i+1\}, \ldots, \{n\}, \{i\})$ for some $1 \leq i < n$. (Every cell except c_i needs to be pushed once to change $\mathbf{s}$ to $\mathbf{s}'$.)

Given two rank-modulation states $\mathbf{s}, \mathbf{s}' \in \mathcal{S}_{n,\lambda}$, we consider how to compute the unweighted rewriting cost $d(\mathbf{s}, \mathbf{s}')$, and how to change $\mathbf{s}$ to $\mathbf{s}'$ with this minimum number of push operations. For the special case $\lambda = 1$, the answer is known [27]: given $\mathbf{s} = (s_1, s_2, \ldots, s_n)$ and $\mathbf{s}' = (s'_1, s'_2, \ldots, s'_n)$, let $\phi : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$ be a bijective map such that for $i = 1, 2, \ldots, n$, we have $s'_i = s_{\phi(i)}$; let r be the minimum integer in $\{1, 2, \ldots, n\}$ such that

$$\phi(r+1) < \phi(r+2) < \cdots < \phi(n);$$

then we have $d(\mathbf{s}, \mathbf{s}') = r$, and the way to change the rank-modulation state from $\mathbf{s}$ to $\mathbf{s}'$ with r push operations is to sequentially pushed the cells with their indices in $s'_r, s'_{r-1}, \ldots, s'_1$ to the top.

For the case $\lambda \geq 2$, we use a tool called *virtual levels*.

Definition 2. *Given a rank-modulation state* $\mathbf{s} = (s_1, s_2, \ldots, s_k) \in \mathcal{S}_{n,\lambda}$, *a "realization" of* $\mathbf{s}$ *is a vector* $(v_1, v_2, \ldots, v_n) \in \mathbb{N}^n$ *that satisfies two conditions: (1)* $\forall$ $1 \leq i \leq k$ *and* $j_1, j_2 \in s_i$, *we have* $v_{j_1} = v_{j_2}$; *(2)* $\forall$ $1 \leq i_1 < i_2 \leq k$, $j_1 \in s_{i_1}$ *and* $j_2 \in s_{i_2}$, *we have* $v_{j_1} > v_{j_2}$. *We call* v_i *the "virtual level" of the cell* c_i, *for*

$i = 1, 2, \ldots, n$.

Definition 3. *Let* $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ *be a realization of* $\mathbf{s} \in \mathcal{S}_{n,\lambda}$*, and let* $\mathbf{v}' = (v'_1, v'_2, \ldots, v'_n)$ *be a realization of* $\mathbf{s}' \in \mathcal{S}_{n,\lambda}$*. The Hamming distance between* $\mathbf{v}$ *and* $\mathbf{v}'$*, denoted by* $H(\mathbf{v}, \mathbf{v}')$*, is* $H(\mathbf{v}, \mathbf{v}') = |\{i \mid 1 \leq i \leq n,\ v_i \neq v'_i\}|$*. And we say "*$\mathbf{v}'$ *dominates* $\mathbf{v}$*" if two conditions are satisfied: (1) for* $i = 1, 2, \ldots, n$*, we have* $v'_i \geq v_i$*; (2) we have* $\{v'_i \mid 1 \leq i \leq n, v'_i \leq \max_{1 \leq j \leq n} v_j\} \subseteq \{v_1, v_2, \ldots, v_n\}$*. We denote "*$\mathbf{v}'$ *dominates* $\mathbf{v}$*" by* $\mathbf{v}' \geq \mathbf{v}$*.*

Lemma 4. *Let* $\lambda \geq 2$*. Let* $\mathbf{s}, \mathbf{s}' \in \mathcal{S}_{n,\lambda}$ *be two rank-modulation states, let* $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ *be a realization of* $\mathbf{s}$*, and let* x *be a non-negative integer. Then,* $\mathbf{s}$ *can be changed into* $\mathbf{s}'$ *by at most* x *push operations if and only if there exists a realization* $\mathbf{v}' = (v'_1, v'_2, \ldots, v'_n)$ *of* $\mathbf{s}'$ *such that* $\mathbf{v}' \geq \mathbf{v}$ *and* $H(\mathbf{v}, \mathbf{v}') \leq x$*.*

Proof. First, assume that $\mathbf{s}$ can be changed into $\mathbf{s}'$ by $y \leq x$ push operations. We will construct a corresponding realization $\mathbf{v}'$ of $\mathbf{s}'$ as follows. Initially, for $i = 1, 2, \ldots, n$, let $v'_i = v_i$. Then for $i = 1, 2, \ldots, y$, if the i-th push operation pushes a cell c_{j_1} to the same rank as another cell c_{j_2}, then assign to v'_{j_1} the value of v'_{j_2}. Otherwise, the i-th push operation pushes a cell c_j to a rank that is higher than all the other $n-1$ cells; in this case, let $z = \max_{1 \leq b \leq n} v'_b$, and we assign to v'_j the value $z + 1$. Then, let $\mathbf{v}' = (v'_1, v'_2, \ldots, v'_n)$. It is simple to see that $\mathbf{v}'$ is a realization of $\mathbf{s}'$ and $\mathbf{v}' \geq \mathbf{v}$. Since at most y cells are pushed, at least $n - y$ cells have the same virtual levels in $\mathbf{v}$ and $\mathbf{v}'$; so we have $H(\mathbf{v}, \mathbf{v}') \leq y \leq x$.

Now consider the other direction. Assume that there exists a realization $\mathbf{v}' = (v'_1, v'_2, \ldots, v'_n)$ of $\mathbf{s}'$ such that $\mathbf{v}' \geq \mathbf{v}$ and $H(\mathbf{v}, \mathbf{v}') \leq x$. We will show how to change $\mathbf{s}$ to $\mathbf{s}'$ with $H(\mathbf{v}, \mathbf{v}')$ push operations. We first partition $\{v'_1, v'_2, \ldots, v'_n\}$ into two

subsets A and B as follows:

$$A = \{v'_i \mid 1 \leq i \leq n, v'_i > \max_{1 \leq j \leq n} v_j\};$$

$$B = \{v'_i \mid 1 \leq i \leq n, v'_i \leq \max_{1 \leq j \leq n} v_j\}.$$

Since $\mathbf{v}' \geq \mathbf{v}$, we know that $B \subseteq \{v_1, v_2, \ldots, v_n\}$. Here B is the set of virtual levels that are retained when we change $\mathbf{s}$ into $\mathbf{s}'$, and A is the set of virtual levels in $\mathbf{v}'$ that are higher than any virtual level in $\mathbf{v}$. For convenience, we shall denote A as $A = \{a_1, a_2, \ldots, a_{|A|}\}$ such that $a_1 < a_2 < \cdots < a_{|A|}$, and denote B as $B = (b_1, b_2, \ldots, b_{|B|})$ such that $b_1 > b_2 > \cdots > b_{|B|}$.

We change the rank-modulation state from $\mathbf{s}$ to $\mathbf{s}'$ as follows. Initially, for $i = 1, 2, \ldots, n$, let the cell c_i have the virtual level v_i. We will push the cells to higher virtual levels, and the rank-modulation state – which is determined by the virtual levels of the n cells – will change accordingly. We push the cells using the following two steps:

1. For $i = 1, 2, \ldots, |A|$, push the cells in $\{c_j \mid 1 \leq j \leq n, v'_j = a_i\}$ to the virtual level a_i.

2. For $i = 1, 2, \ldots, |B|$, push the cells in $\{c_j \mid 1 \leq j \leq n, v_j < v'_j = b_i\}$ to the virtual level b_i.

During the above two steps, we will use the following method to make sure that for $i = 1, 2, \ldots, |B|$, there is always at least one cell of the virtual level b_i:

- When we are to push a cell c_i from the virtual level $j_1 \in B$ to $j_2 > j_1$, if c_i is the only cell of virtual level j_1 at that moment, then before pushing c_i, we first push a cell in $\{c_z \mid 1 \leq z \leq n, v'_z = j_1\}$ to the virtual level j_1. (Note that

if *that* cell is also the only cell of its own virtual level at that moment, then the same rule applies. So there can be a chain reaction of cell pushing of this type. But this chain reaction will stop somewhere because the virtual level of the concerned cell keeps decreasing.)

In the above process, we push every cell at most once.

When the above process ends, the cells have virtual levels $(v'_1, v'_2, \ldots, v'_n)$, which is a realization of $\mathbf{s}'$. A cell c_i $(1 \leq i \leq n)$ is pushed if and only if $v_i \neq v'_i$; and if it is pushed, it is pushed directly to the virtual level v'_i. So the number of push operations equals $H(\mathbf{v}, \mathbf{v}')$. We now show that these $H(\mathbf{v}, \mathbf{v}')$ push operations are all valid operations for the rank-modulation states. Step 1) consists of the "push-to-top" operations, and we sequentially push the cells to higher and higher ranks; clearly, the number of cells at the virtual level a_i (for $1 \leq i \leq |A|$) is never more than λ at any moment. Step 2) consists of the operations that push a cell to a higher and existing rank; and since we process the virtual levels $b_1, b_2, \ldots, b_{|B|}$ sequentially (from high to low), when we process the virtual level b_i (for $1 \leq i \leq |B|$), all the cells that are originally at level b_i have already been pushed up; so as we push cells from below into the level b_i, there will be no more than λ cells in that level. So we have changed $\mathbf{s}$ into $\mathbf{s}'$ with $H(\mathbf{v}, \mathbf{v}') \leq x$ valid push operations. □

Theorem 5. *Let $\lambda \geq 2$. Let $\mathbf{s} = (s_1, s_2, \ldots, s_k) \in \mathcal{S}_{n,\lambda}$ and $\mathbf{s}' = (s'_1, s'_2, \ldots, s'_{k'}) \in \mathcal{S}_{n,\lambda}$ be two rank-modulation states, let $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ be a realization of $\mathbf{s}$, and define V as $V = \{\mathbf{u} \mid \mathbf{u}$ is a realization of $\mathbf{s}'$, $\mathbf{u} \geq \mathbf{v}\}$. Then we have*

$$d(\mathbf{s}, \mathbf{s}') = \min_{\mathbf{u} \in V} H(\mathbf{v}, \mathbf{u}).$$

Furthermore, define $\mathbf{v}' = (v'_1, v'_2, \ldots, v'_n)$ as follows:

1. *Let $h_{k'} = \max_{j \in s'_{k'}} v_j$.*

$\forall\ i \in s'_{k'}$, *let* $v'_i = h_{k'}$.

2. *For* $i_1 = k'-1, k'-2, \ldots, 1$, *do:*

   - *If* $\max_{j \in s'_{i_1}} v_j > h_{i_1+1}$, *then let*

     $h_{i_1} = \max_{j \in s'_{i_1}} v_j$;

     if $\max_{j \in s'_{i_1}} v_j \leq h_{i_1+1} < \max_{1 \leq j \leq n} v_j$, *then let*

     $h_{i_1} = \min\{v_j \mid 1 \leq j \leq n, v_j > h_{i_1+1}\}$;

     if $\max_{j \in s'_{i_1}} v_j \leq h_{i_1+1}$ *and* $h_{i_1+1} \geq \max_{1 \leq j \leq n} v_j$, *then let*

     $h_{i_1} = h_{i_1+1} + 1$.

   - $\forall\ i_2 \in s'_{i_1}$, *let* $v'_{i_2} = h_{i_1}$.

Then we have

$$\mathbf{v}' \in V \text{ and } H(\mathbf{v}, \mathbf{v}') = \min_{\mathbf{u} \in V} H(\mathbf{v}, \mathbf{u})$$

Proof. Lemma 4 leads to $d(\mathbf{s}, \mathbf{s}') = \min_{\mathbf{u} \in V} H(\mathbf{v}, \mathbf{u})$. When we assign values to $(v'_1, v'_2, \ldots, v'_n)$ (which are virtual levels for the n cells corresponding to the rank-modulation state $\mathbf{s}'$), we are sequentially assigning virtual levels to the cells with indices in $s'_{k'}, s'_{k'-1}, \ldots, s'_1$; and for $i = k', k'-1, \ldots, 1$, we give the cells with indices in s'_i a virtual level that is as small as possible, as long as the condition $\mathbf{v}' \in V$ is satisfied. A proof by induction can show that compared to all the realizations of $\mathbf{s}'$ in V, here each h_i $(1 \leq i \leq k')$ – and therefore each virtual level v'_i $(1 \leq i \leq n)$ – is individually minimized, and a cell is pushed only when necessary. (Since the cells are pushed only upward, minimizing h_i is a greedy and optimal approach for minimizing $h_{i-1}, h_{i-2}, \ldots, h_1$ and for minimizing the number of cells that need to be pushed.) So $H(\mathbf{v}, \mathbf{v}') = \min_{\mathbf{u} \in V} H(\mathbf{v}, \mathbf{u})$. □

Theorem 5 shows how to find the realization $\mathbf{v}'$ for $\mathbf{s}'$ such that $\mathbf{v}'$ dominates $\mathbf{v}$ (the realization of $\mathbf{s}$) and $H(\mathbf{v}, \mathbf{v}') = d(\mathbf{s}, \mathbf{s}')$. The proof of Lemma 4 shows given such

a realization $\mathbf{v}$, how to change the rank-modulation state from $\mathbf{s}$ to $\mathbf{s}'$ with $d(\mathbf{s}, \mathbf{s}')$ push operations. By combining them, we can not only compute $d(\mathbf{s}, \mathbf{s}')$, but also transform $\mathbf{s}$ to $\mathbf{s}'$ with the minimum unweighted rewriting cost. We show an example below.

Example 6. *Suppose* $\lambda = 2$, $n = 8$, $\mathbf{s} = (\{2,3\},\{7\},\{4,5\},\ \{1,8\},\{6\})$, $\mathbf{s}' = (\{2,3\},\{4\},\{1\},\{7,8\},\{5,6\})$. *We let* $\mathbf{v} = (2,5,5,3,3,1,4,2)$ *be a realization of* $\mathbf{s}$. *(See Figure 2.2.) Then by Theorem 5, we get the realization* $\mathbf{v}' = (5,7,7,6,3,3,4,4)$ *of* $\mathbf{s}'$. *(It can be seen that* $\mathbf{v}' \geq \mathbf{v}$.*) So we get* $d(\mathbf{s}, \mathbf{s}') = H(\mathbf{v}, \mathbf{v}') = 6$. *Then by the steps specified in the proof of Lemma 4, we get the 6 push operations that change* $\mathbf{s}$ *into* $\mathbf{s}'$. *(See Figure 2.2, where the push operations are shown as arrows, and the numbers beside arrows represent their order.)*

2.4 Sizes of Spheres

For a rank-modulation state $\mathbf{s} \in \mathcal{S}_{n,\lambda}$ and an unweighted rewriting cost $r \geq 0$, we define the *sphere of unweighted radius* r *centered at* $\mathbf{s}$ as

$$\theta(\mathbf{s}, r) \triangleq \{\mathbf{u} \in \mathcal{S}_{n,\lambda} \mid d(\mathbf{s}, \mathbf{u}) = r\}$$

and define the *ball of unweighted radius* r *centered at* $\mathbf{s}$ as

$$\beta(\mathbf{s}, r) \triangleq \{\mathbf{u} \in \mathcal{S}_{n,\lambda} \mid d(\mathbf{s}, \mathbf{u}) \leq r\}$$

Clearly, $|\beta(\mathbf{s}, r)| = \sum_{i=0}^{r} |\theta(\mathbf{s}, i)|$. Knowing the sizes of spheres and balls is useful for analyzing the performance of rewriting. For example, when the states in $\mathcal{S}_{n,\lambda}$ are used to represent data of the alphabet $\mathcal{L}$, if the rank-modulation state is currently $\mathbf{s} \in \mathcal{S}_{n,\lambda}$, for the next rewrite, the unweighted rewriting cost in the worst case is at

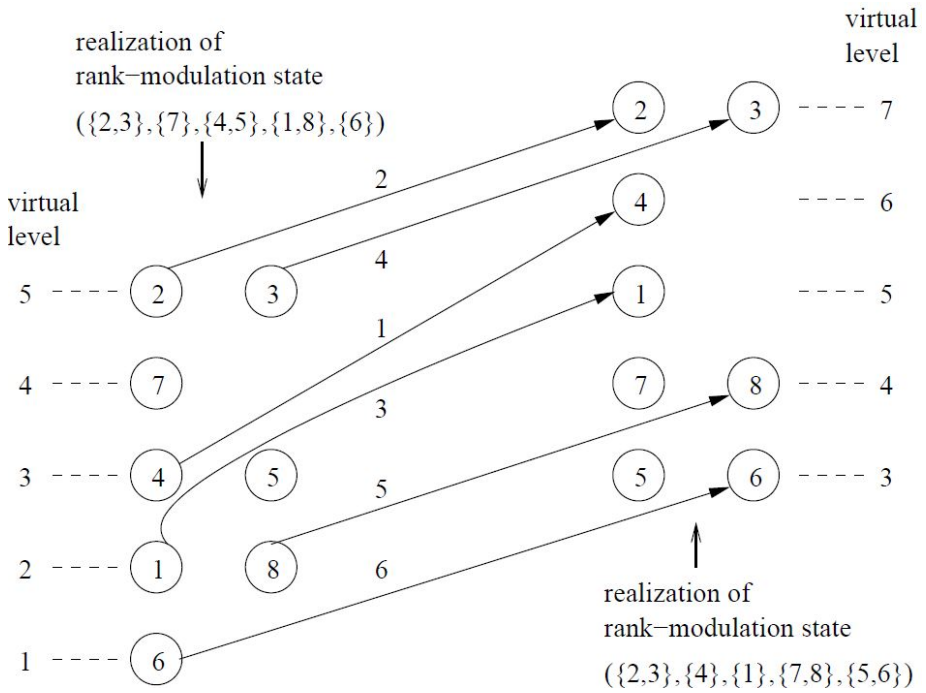


Figure 2.2: Change rank-modulation state from $\mathbf{s}$ to $\mathbf{s}'$ with $d(\mathbf{s}, \mathbf{s}')$ pushes.

least $\min\{r \mid r \geq 0, |\beta(\mathbf{s}, r)| \geq |\mathcal{L}|\}$.

We show how to compute $|\theta(\mathbf{s}, r)|$ for $\mathbf{s} \in \mathcal{S}_{n,\lambda}$ and $0 \leq r \leq n-1$. If $\lambda = 1$, we have

$$|\theta(\mathbf{s}, r)| = \frac{n!}{(n-r)!} - \frac{n!}{(n-r+1)!}$$

for $1 \leq r \leq n-1$ and $|\theta(\mathbf{s}, 0)| = 1$ [27]. So in the following, we consider $\lambda \geq 2$. Fix a realization $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ for $\mathbf{s} = (s_1, s_2, \ldots, s_k)$ – say the realization where the n cells have virtual levels from 1 to k – and we see that for any $\mathbf{s}' \in \mathcal{S}_{n,\lambda}$, Theorem 5 finds a unique realization $\mathbf{v}' = (v'_1, v'_2, \ldots, v'_n)$ for $\mathbf{s}'$ such that $\mathbf{v}' \geq \mathbf{v}$, $H(\mathbf{v}, \mathbf{v}') = d(\mathbf{s}, \mathbf{s}')$ and every virtual level v'_i $(1 \leq i \leq n)$ is minimized. So to compute $|\theta(\mathbf{s}, r)|$, the number of states in the sphere $\theta(\mathbf{s}, r)$, we can equivalently compute the number of such unique realizations (of the states in $\theta(\mathbf{s}, r)$), because they have a one to one correspondence.

Let $\sigma_1, \sigma_2, \ldots, \sigma_\kappa$ and X be $\kappa+1$ mutually disjoint sets of cells, where $1 \leq |\sigma_i| \leq \lambda$ for $1 \leq i \leq \kappa$ and $|X| = x \in \{0, 1, \ldots, n-1\}$. For $i = 1, \ldots, \kappa$, we assign the virtual level $\kappa + 1 - i$ to the cells in the set σ_i. Let $\delta \in \{0, 1, \ldots, n-1\}$, $t \in \{1, 2, \ldots, \lambda\}$, $\gamma \in \{x, x+1, \ldots, n-1\}$ and $tag \in \{0, 1\}$ be given parameters. Let $\mathcal{R}$ denote the set of realizations (that is, assignments of virtual levels to the $x + \sum_{i=1}^{\kappa} |\sigma_i|$ cells) that we can change this current realization into, given the following constraints:

1. We obtain a realization in $\mathcal{R}$ by pushing $\gamma - x$ cells in $\cup_{i=1}^{\kappa} \sigma_i$ to higher virtual levels, and by assigning the x cells in X to the virtual levels between 1 and $\kappa + \delta$. Every cell is pushed or assigned at most once. For the realization in $\mathcal{R}$, every virtual level has at most λ cells.

2. For a realization in $\mathcal{R}$, the maximum virtual level that has a cell is level $\kappa + \delta$, and exactly t cells are in that virtual level $\kappa + \delta$.

3. For a realization in $\mathcal{R}$, if a cell in $\cup_{i=1}^{\kappa} \sigma_i$ is pushed to a level $j \in \{2, 3, \ldots, \kappa+\delta\}$, or if a cell in X is assigned to a level $j \in \{2, 3, \ldots, \kappa+\delta\}$, then for this realization in $\mathcal{R}$, either some cell is in the virtual level $j - 1$, or $2 \leq j \leq \kappa$ and some cell in $\sigma_{\kappa+1-j}$ is in the level j.

4. If $tag = 1$, then no cell in X can be assigned to the virtual level 1 unless for this realization in $\mathcal{R}$, some cell in σ_κ is in the virtual level 1.

We use $f(|\sigma_1|, |\sigma_2|, \ldots, |\sigma_\kappa|; x; \delta; t; \gamma; tag)$ to denote the cardinality of $\mathcal{R}$. We can see that the sphere size

$$|\theta(\mathbf{s}, r)| = \sum_{\delta=0}^{r} \sum_{t=1}^{\lambda} f(|s_1|, |s_2|, \ldots, |s_k|; 0; \delta; t; r; 0).$$

We show how to use recursion to compute the value of $f(|\sigma_1|, |\sigma_2|, \ldots, |\sigma_\kappa|; x; \delta; t; \gamma; tag)$. For simplicity, we only introduce the main recursion, and skip introducing the values

of $f(|\sigma_1|, |\sigma_2|, \ldots, |\sigma_\kappa|; x; \delta; t; \gamma; tag)$ for the boundary cases. (The boundary values can be obtained easily.)

To change the given realization to a realization in $\mathcal{R}$, say that we push y_1 cells in σ_κ to the maximum virtual level $k+\delta$, push y_2 cells in σ_κ to the virtual levels $2, 3, \ldots, k+\delta-1$, and assign y_3 cells in X to the virtual level 1. Note that once y_1, y_2, y_3 are fixed, the number of cells in level 1 becomes fixed, and we do not need to consider it furthermore. So we get the recursion:

- If $tag = 0$, then let

$$P_1 \triangleq \{(y_1,y_2,y_3) \in \mathbb{Z}^3 \mid 0 \le y_1 < t; 0 \le y_2 \le |\sigma_\kappa|; 0 \le y_3 \le \min\{x, \lambda - |\sigma_\kappa| + y_1 + y_2\};$$
$$\text{either } \text{“}y_1 + y_2 < |\sigma_\kappa| \text{”} \text{ or } \text{“}y_1 + y_2 = |\sigma_\kappa| \text{ and } y_3 > 0\text{”}\},$$
$$P_2 \triangleq \{(y_1,y_2,y_3) \in \mathbb{Z}^3 \mid 0 \le y_1 \le \min\{t-1, |\sigma_\kappa|\}; y_2 = |\sigma_\kappa| - y_1; y_3 = 0\},$$
$$P_3 \triangleq \{(y_1,y_2,y_3) \in \mathbb{Z}^3 \mid y_1 = t; 0 \le y_2 \le |\sigma_\kappa|; 0 \le y_3 \le \min\{x, \lambda - |\sigma_\kappa| + y_1 + y_2\};$$
$$\text{either } \text{“}y_1 + y_2 < |\sigma_\kappa| \text{”} \text{ or } \text{“}y_1 + y_2 = |\sigma_\kappa| \text{ and } y_3 > 0\text{”}\},$$
$$P_4 \triangleq \{(y_1,y_2,y_3) \in \mathbb{Z}^3 \mid y_1 = t; y_2 = |\sigma_\kappa| - t \ge 0; y_3 = 0\}$$

If $tag = 1$, then let

$$P_1 \triangleq \{(y_1,y_2,y_3) \in \mathbb{Z}^3 \mid 0 \le y_1 < t; 0 \le y_2 < |\sigma_\kappa| - y_1; 0 \le y_3 \le \min\{x, \lambda - |\sigma_\kappa| + y_1 + y_2\}\},$$

$$P_2 \triangleq \{(y_1,y_2,y_3) \in \mathbb{Z}^3 \mid 0 \le y_1 \le \min\{t-1, |\sigma_\kappa|\}; y_2 = |\sigma_\kappa| - y_1; y_3 = 0\},$$

$$P_3 \triangleq \{(y_1,y_2,y_3) \in \mathbb{Z}^3 \mid y_1 = t; 0 \le y_2 < |\sigma_\kappa| - t; 0 \le y_3 \le \min\{x, \lambda - |\sigma_\kappa| + y_1 + y_2\}\},$$

$$P_4 \triangleq \{(y_1,y_2,y_3) \in \mathbb{Z}^3 \mid y_1 = t; y_2 = |\sigma_\kappa| - t \ge 0; y_3 = 0\}$$

- We have

$$f(|\sigma_1|, |\sigma_2|, \ldots, |\sigma_\kappa|; x; \delta; t; \gamma; tag) = \sum_{(y_1,y_2,y_3)\in P_1} \binom{|\sigma_\kappa|}{y_1}\binom{|\sigma_\kappa|-y_1}{y_2}\binom{x}{y_3} f(|\sigma_1|, |\sigma_2|, \cdots, |\sigma_{\kappa-1}|; x+y_2-y_3; \delta; t-y_1; \gamma-y_1-y_3; 0) + \sum_{(y_1,y_2,y_3)\in P_2} \binom{|\sigma_\kappa|}{y_1}\binom{|\sigma_\kappa|-y_1}{y_2}\binom{x}{y_3} f(|\sigma_1|, |\sigma_2|, \cdots, |\sigma_{\kappa-1}|; x+y_2; \delta; t-y_1; \gamma-y_1; 1) + \sum_{(y_1,y_2,y_3)\in P_3} \binom{|\sigma_\kappa|}{y_1}\binom{|\sigma_\kappa|-y_1}{y_2}\binom{x}{y_3} \sum_{1\le z\le\lambda} f(|\sigma_1|, |\sigma_2|, \cdots, |\sigma_{\kappa-1}|; x+y_2-y_3; \delta-1; z; \gamma-y_1-y_3; 0) + \sum_{(y_1,y_2,y_3)\in P_4} \binom{|\sigma_\kappa|}{y_1}\binom{|\sigma_\kappa|-y_1}{y_2}\binom{x}{y_3} \sum_{1\le z\le\lambda} f(|\sigma_1|, |\sigma_2|, \cdots, |\sigma_{\kappa-1}|; x+y_2; \delta-1; z; \gamma-y_1; 1).$$

Given any $\mathbf{s} \in \mathcal{S}_{n,\lambda}$ and $r \le n-1$, the time complexity of computing the sphere size $|\theta(\mathbf{s}, r)|$ using the above recursion is $O(n^4\lambda^5)$.

Theorem 7. *The above recursion correctly computes* $|\theta(\mathbf{s}, r)|$.

Proof. In order to compute $|\theta(\mathbf{s}, r)|$, we enumerate all the possible ways to push r cells by the recursion. The recursion is processed in the order of cells' virtual levels: the cells in $\sigma_{\kappa+1-i}$, $1 \le i \le \kappa$ with virtual level i are processed in the i-th round. If a cell is pushed, it is only pushed once. We can get four disjoint cases after pushing $y_1 + y_2$ cells in σ_κ and assigning y_3 cells in X to some virtual levels:

1. P_1: σ_κ is not empty, and at least one cell with the maximum virtual level $k+\delta$ is not coming from σ_κ.

2. P_2: σ_κ is empty (no cell is assigned to the same virtual level as cells in σ_κ originally have), and at least one cell with the maximum virtual level $k+\delta$ is not coming from σ_κ.

3. P_3: σ_κ is not empty, and all the cells with the maximum virtual level $k+\delta$ are coming from σ_κ.

4. P_4: σ_κ is empty (no cell is assigned to the same virtual level as cells in σ_κ originally have), and all the cells with the maximum virtual level $k+\delta$ are coming from σ_κ.

For each case, we set the appropriate parameters in the above recursive function. Since it covers all the possible ways to push r cells, the result from the recursion is no less than $|\theta(\mathbf{s}, r)|$.

On the other side, we need to prove all the ways to push cells getting from the recursion are unique and valid. It is easily to see that all the push operation sequences are unique because the above four cases are disjoint and each cell is processed only once. In order to prove they are valid, we need to show 1) at each step, no more than λ cells are with the same virtual level; 2) If a cell is pushed, it is pushed to the smallest virtual level. The parameters' upper bounds in P_1, P_2, P_3, P_4 as well as the recursive function guarantee that the number of cells at each virtual level is never more than λ at any moment. The parameter tag is used to assign each cell to the smallest virtual level in its realization. For the case P_2 and P_4, no cell is assigned to the same virtual level as cells in σ_κ originally have in the realization, we label tag to 1. Then during the next round recursion, (when we deal with the cells in $\sigma_{\kappa-1}$), no cell in X can be assigned to the virtual level as cells in $\sigma_{\kappa-1}$ originally have, unless in this realization, some cells in $\sigma_{\kappa-1}$ are not pushed up. Otherwise, the cells in X can be assigned to virtual level as cells in σ_κ originally have, which is lower than virtual levels in $\sigma_{\kappa-1}$, when we deal with the cells in σ_κ to get the state $\mathbf{s}'$. Therefore, the tag parameter guarantees that each cell is assigned to the smallest virtual level to reach state $\mathbf{s}'$, such that $d(\mathbf{s}, \mathbf{s}') = r$. □

2.5 Weighted Rewriting Cost

We have studied the unweighted rewriting cost, where every push operation is considered to have cost one. In practice, however, the operations can have different cost values: a push operation that increases the cell level less is more preferable than a push operation that increases the cell level more. So in this section, we present

the definition of *weighted rewriting cost*, which measures the cost of push operations based on how much they increase the cell levels.

As a combinatorial definition, we use the help of virtual levels. Let $\mathbf{s} = (s_1, s_2, \ldots, s_k) \in \mathcal{S}_{n,\lambda}$ and $\mathbf{s}' \in \mathcal{S}_{n,\lambda}$ be two rank-modulation states. Let $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ be the unique realization of $\mathbf{s}$ such that $\{v_1, v_2, \ldots, v_n\} = \{1, 2, \ldots, k\}$. Let $V \triangleq \{\mathbf{u} \mid \mathbf{u}$ is a realization of $\mathbf{s}'$, $\mathbf{u} \geq \mathbf{v}\}$. By the previous analysis, we know that a sequence of push operations that changes the rank-modulation state from $\mathbf{s}$ to $\mathbf{s}'$ also changes the realization from $\mathbf{v}$ to some $\mathbf{u} \in V$ (and vice versa). Virtual levels are a reasonable simplification of real cell levels. So we define the weighted rewriting cost of changing $\mathbf{s}$ into $\mathbf{s}'$ as

$$w(\mathbf{s}, \mathbf{s}') = \min_{(u_1, u_2, \ldots, u_n) \in V} \sum_{i=1}^{n} (u_i - v_i).$$

Let $\mathbf{v}' = (v'_1, v'_2, \ldots, v'_n)$ be the unique realization of $\mathbf{s}'$ that is generated by Theorem 5. It has been shown that $\mathbf{v}'$ minimizes the virtual level of every cell; so we have

$$w(\mathbf{s}, \mathbf{s}') = \sum_{i=1}^{n} (v'_i - v_i) = \sum_{i=1}^{n} \min_{(u_1, \ldots, u_n) \in V} (u_i - v_i).$$

And it is not hard to see that

$$\max_{\mathbf{s}, \mathbf{s}' \in \mathcal{S}_{n,\lambda}} w(\mathbf{s}, \mathbf{s}') = n(n-1).$$

Given a state $\mathbf{s} \in \mathcal{S}_{n,\lambda}$ and an integer $r \geq 0$, we can define the *sphere of weighted radius r centered at* $\mathbf{s}$ as

$$\Theta(\mathbf{s}, r) \triangleq \{\mathbf{u} \in \mathcal{S}_{n,\lambda} \mid w(\mathbf{s}, \mathbf{u}) = r\}$$

The sphere size, $|\Theta(\mathbf{s}, r)|$, can be computed with a similar recursion as the one in the previous section.

Like the previous section, we also define $\sigma_1, \sigma_2, \ldots, \sigma_\kappa$ and X to be $\kappa+1$ mutually disjoint sets of cells, where $1 \leq |\sigma_i| \leq \lambda$ for $1 \leq i \leq \kappa$ and $|X| = x \in \{0, 1, \ldots, n-1\}$. And $\delta \in \{0, 1, \ldots, n-1\}$, $t \in \{1, 2, \ldots, \lambda\}$, $\gamma \in \{x, x+1, \ldots, n-1\}$ and $tag \in \{0, 1\}$ are also the given parameters. Let $\mathcal{R}$ denote the set of realizations (that is, assignments of virtual levels to the $x + \sum_{i=1}^{\kappa} |\sigma_i|$ cells) that we can change this current realization into, given the following constraints:

1. We obtain a realization in $\mathcal{R}$ by pushing some cells in $\cup_{i=1}^{\kappa} \sigma_i$ to higher virtual levels, and by assigning the x cells in X to the virtual levels between 1 and $k + \delta$, with the summation of cells' virtual level increased by γ. Every cell is pushed or assigned at most once. For the realization in $\mathcal{R}$, every virtual level has at most λ cells.

2. For a realization in $\mathcal{R}$, the maximum virtual level that has a cell is level $k + \delta$, and exactly t cells are in that virtual level $k + \delta$.

3. For a realization in $\mathcal{R}$, if a cell in $\cup_{i=1}^{\kappa} \sigma_i$ is pushed to a level $j \in \{2, 3, \ldots, k+\delta\}$, or if a cell in X is assigned to a level $j \in \{2, 3, \ldots, k+\delta\}$, then for this realization in $\mathcal{R}$, either some cell is in the virtual level $j - 1$, or $2 \leq j \leq \kappa$ and some cell in $\sigma_{\kappa+1-j}$ is in the level j.

4. If $tag = 1$, then no cell in X can be assigned to the virtual level 1 unless for this realization in $\mathcal{R}$, some cell in σ_κ is in the virtual level 1.

We use $g(|\sigma_1|, |\sigma_2|, \ldots, |\sigma_\kappa|; x; \delta; t; \gamma; tag)$ to denote the cardinality of $\mathcal{R}$. We can

see that the sphere size

$$|\Theta(\mathbf{s}, r)| = \sum_{\delta=0}^{\min\{n-1,r\}} \sum_{t=1}^{\lambda} g(|s_1|, |s_2|, \ldots, |s_k|; 0; \delta; t; r; 0).$$

We use the same way to change the given realization to a realization in $\mathcal{R}$ and define the set P_1, P_2, P_3, P_4 as the previous section. We have

$g(|\sigma_1|, |\sigma_2|, \ldots, |\sigma_\kappa|; x; \delta; t; \gamma; tag) = \sum_{(y_1,y_2,y_3)\in P_1} \binom{|\sigma_\kappa|}{y_1}\binom{|\sigma_\kappa|-y_1}{y_2}\binom{x}{y_3} g(|\sigma_1|, |\sigma_2|, \cdots, |\sigma_{\kappa-1}|; x+y_2-y_3; \delta; t-y_1; \gamma-y_1(\delta-1+\kappa)-y_2-x+y_3; 0) + \sum_{(y_1,y_2,y_3)\in P_2} \binom{|\sigma_\kappa|}{y_1}\binom{|\sigma_\kappa|-y_1}{y_2}\binom{x}{y_3} g(|\sigma_1|, |\sigma_2|, \cdots, |\sigma_{\kappa-1}|; x+y_2; \delta; t-y_1; \gamma-y_1(\delta-1+\kappa)-y_2-x; 1) + \sum_{(y_1,y_2,y_3)\in P_3} \binom{|\sigma_\kappa|}{y_1}\binom{|\sigma_\kappa|-y_1}{y_2}\binom{x}{y_3} \sum_{1\le z\le\lambda} g(|\sigma_1|, |\sigma_2|, \cdots, |\sigma_{\kappa-1}|; x+y_2-y_3; \delta-1; z; \gamma-y_1(\delta-1+\kappa)-y_2-x+y_3; 0) + \sum_{(y_1,y_2,y_3)\in P_4} \binom{|\sigma_\kappa|}{y_1}\binom{|\sigma_\kappa|-y_1}{y_2}\binom{x}{y_3} \sum_{1\le z\le\lambda} g(|\sigma_1|, |\sigma_2|, \cdots, |\sigma_{\kappa-1}|; x+y_2; \delta-1; z; \gamma-y_1(\delta-1+\kappa)-y_2-x; 1).$

Theorem 8. *The above recursion correctly computes* $|\Theta(\mathbf{s}, r)|$.

Proof. The recursion is similar to the unweighted case, except for the parameter γ. By pushing y_1 cells in σ_κ to virtual level $k+\delta$, y_2 cells in σ_κ to some virtual level less than $k+\delta$ and assigning y_3 cells in X to virtual level $k+1-\kappa$, the cells' virtual level is totally increased by $y_1(\delta-1+\kappa)+y_2+x-y_3$ or $y_1(\delta-1+\kappa)+y_2+x$ (when $y_3=0$). Therefore, the parameter γ is set as $\gamma-y_1(\delta-1+\kappa)-y_2-x+y_3$ or $\gamma-y_1(\delta-1+\kappa)-y_2-x$ in the recursive function $g(\cdot)$, which is equal to the remaining weighted distance from the current state to the final state. The other part is the same as the proof of Theorem 7. □

3. EXPLOITING HALF-WITS: SMARTER STORAGE FOR LOW-POWER DEVICES

The high voltage requirements of on-chip flash memory is a barrier to reducing the total energy consumption of low-power devices. This work examines the main factors affecting the behavior of flash memory at low voltage. Based on our observations of flash memory behavior at low voltage, we proposed three storage schemes to enable reliable storage on flash memory. The first scheme, *in-place writes*, makes attempts at *write time* to store a value correctly in the given memory address. The second scheme, *multiple-place writes*, tries to decrease the probability of error by making attempts at both *write time* and *read time*. This method stores data in more than one location hoping that the data will be stored correctly in at least one of these locations. The third scheme is a hybrid error-correcting code combining Reed-Solomon (RS) [44] and Berger [6] codes. The Berger code detects asymmetric errors caused by the low write voltage. Given the approximate locations of errors, which are determined by the Berger code, the RS code efficiently recovers the originally stored data. Our evaluation shows that in-place writes can save 34% of energy consumption for a sensing workload on the MSP430 microcontroller.

3.1 Storage on Low-Power Devices: Limitations and Challenges

Billions of microcontrollers appear in embedded systems ranging from thermostats and utility meters to tollway payment transponders and pacemakers. Recently years have witnessed a proliferation of low-power embedded devices [4, 8, 37], many of which use on-chip flash memory for storage.

The relatively high voltage requirement of flash memory (Table 3.1) introduces challenges for energy-efficient designs aiming to maximize the system's effective life-

time. Instrumenting the system to operate at a fixed low voltage v_l is one way to reduce power consumption; however, achieving *consistently correct* results for flash writes are guaranteed only if v_l is higher than a manufacturer-specified threshold. Moreover, in energy-limited devices that cannot provide a constant supply voltage, scenarios may arise in which the flash memory is the only part of the circuit whose operating requirements are not met. In such cases, applications can expect normal operation when they are not performing flash writes and unpredictable behavior when they are.

Table 3.1: CPU vs flash memory voltage requirements

Microcontroller	CPU Min. voltage	Flash write Min. voltage
TI MSP430 [24]	1.8V	2.2V or 2.7V
PIC32M [40]	2.3V	3.0V
ATmega128L [53]	2.7V	4.5V

Because embedded flash memory typically shares a common voltage supply with the CPU (separate power rails are cost prohibitive), a single voltage must be chosen that satisfies different components with different minimum voltage requirements. Current embedded systems address the voltage limitations of flash memory in one of the following ways:

1. A system can choose a high supply voltage sufficient for both reliable writes to flash memory and reliable CPU operation. This is a common choice for embedded systems with on-chip flash memory, but causes the CPU to consume more energy than necessary. For example, the TI MSP430F2131 microcontroller [24]

in active mode consumes almost double the power when operating at 2.2V instead of 1.8V. Its on board flash memory requires 2.2V for reliable writes to flash memory.

2. A system can choose a low supply voltage sufficient for CPU operation, but insufficient for reliable writes to flash memory. This choice allows the energy source to last longer and for the CPU to compute more efficiently. An example of such a system is the Intel WISP [48], a batteryless RFID tag that sets its operating voltage to 1.8V—below its onboard flash memory's 2.2V specified minimum—to save power. Flash memory cannot be written on this device. The microcontroller could use a low-power wireless interface (e.g., RF backscatter) to store data remotely. Such an approach, however, raises privacy as well as performance concerns [46].

3. A system can modify hardware to enable dynamic voltage scaling. This approach requires additional analog circuitry such as voltage regulators and GPIO-controlled switches. Because many embedded systems are extremely cost sensitive, this choice is unattractive for high-volume manufacturing with low per-unit profit margins. An additional 50 cent part on a thermostat control can be cost prohibitive. Moreover, small changes may necessitate a new PCB layout—upsetting the delicate supply chain and invalidating stocked inventories of already fabricated PCBs.

Approach Our approach reduces the operating voltage of the microcontroller to a point at which the resulting power savings of the CPU portion of the workload exceeds the power cost of the algorithms for ensuring reliable writes (Figure 3.1). Our low-power storage scheme benefits from the accumulative property of flash memory by repeating writes to the same cell. Each write operation will increase the chance

of success by forcing some number of state transitions. That is, a failed write is still progress. The technique requires minimal or no hardware modification and also allows for RFID-scale and small-scale energy harvesting devices to better exploit capacitors as power supplies. The capacitor provides finite energy and therefore the voltage decays exponentially. The long tail of the curve provides insufficient voltage for conventional writes to flash memory, but it is sufficient for reliable storage with our techniques.

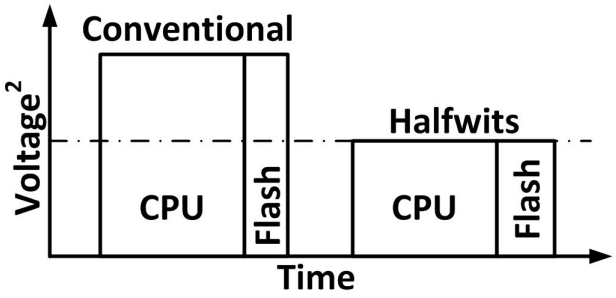


Figure 3.1: Operating at a lower voltage and tolerating errors instead of the conventional case of choosing the highest minimum voltage requirement may help decrease energy consumption. Considering that Energy = voltage^2×time/resistance, decreasing voltage decreases the energy consumption quadratically.

Of wits and half-wits In 1982, Rivest and Shamir introduced the notion of write-once bits (Wits) in the context of coding theory to make write-once storage behave like read-write storage [45]. Bits in flash memory behave like wits because a programmed bit cannot be reprogrammed without calling an energy-intensive erase operation to a block of memory much larger than a single write. We coin the term *Half-Wits* to refer to wits used in a manner inconsistent with a manufacturer's specifications, resulting in stochastic behavior. Half-Wits in this work are wits of flash memory used below the recommended supply voltage.

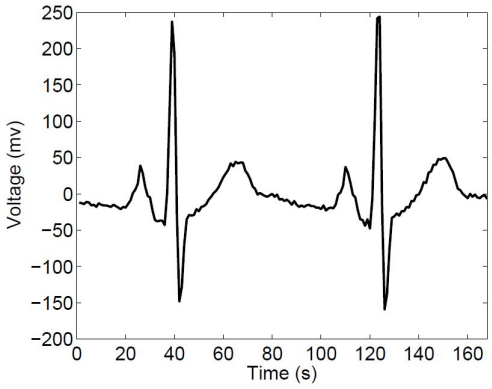
In examining error rates at low voltage and constructing a system that provides

reliable storage despite errors, our work suggests that it is appropriate to relax previously assumed constraints and reexamine the costly digital abstractions layered above on-chip flash memory.

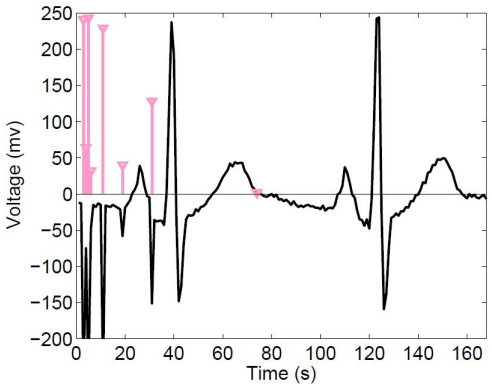
3.2 Behavior of Storage on Half-Wits

Before we can design effective coding algorithms, we must first understand the behavior of errors on Half-Wits. By tolerating a lower voltage, an energy-limited embedded device can decrease its power consumption and therefore extend its lifetime on a finite energy supply. The minimum operating voltage of embedded devices that use non-volatile on-chip storage is usually determined by the requirements of flash memory. For example, the TI MSP430 microcontroller can operate at 1.8V, but its nominal minimum voltage for flash writing and erasure is 2.2V (Table 3.1). Increasing operating voltage from 1.8V to 2.2V causes the CPU to draw about 50% more power without commensurate gain in clock speed because of the voltage squaring effect.

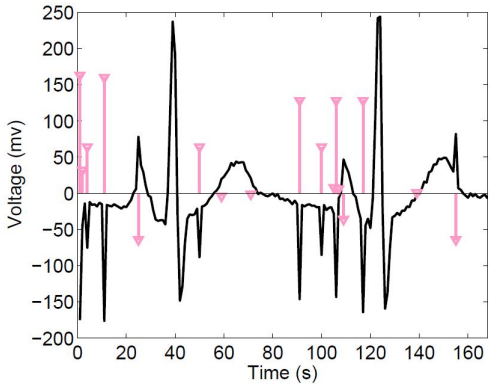
The drawback of lowering voltage below flash memory requirements in order to save power is the extra work necessary to ensure reliable writes to flash memory. Figure 3.2 shows the result of running a MSP430F2131 at three different voltages—all lower than the nominal minimum for flash writes—to store electrocardiogram (ECG) data samples from the PhysioNet database [21] in flash memory. Many medical sensor networks [38, 52] that provide ECG measurements are energy-limited and use on-chip flash memory as primary storage.



(a) Writes at 2.0V



(b) Writes at 1.9V



(c) Writes at 1.8V

Figure 3.2: As operating voltage decreases, flash-write errors increase. (a) shows an original ECG signal correctly stored at 2.0V (despite operating below the recommended threshold). As the voltage decreases in (b) and further in (c), erroneous writes (light-colored spikes, height varying according to the magnitude of the error) become more common. The back line shows the reconstructed signal that includes the errors.

These graphs support the intuition that flash writes may not be error-free at low voltages and that there exist voltage levels below the minimum recommended voltage at which flash writes function correctly. To investigate the behavior of flash memory at low voltage and determine the factors influencing the error rate, we performed experiments on an automated testbed designed by Salajegheh [47].

3.2.1 Experimental Methodology

We use a consistent experimental setup for all of the experiments in this work. Our choice of test platform is a TI MSP430 microcontroller with on-chip flash memory. More specifically, we tested two types of TI microcontrollers: MSP430F2131 and MSP430F1232. The MSP430 is common in low-power embedded applications; we note especially its use in sensor motes [43] and RFID-scale batteryless devices [48]. In our setup, an MSP430 microcontroller runs a test program that involves both computation and flash operation. We power the microcontroller with an external power supply held steady at a voltage below the nominal minimum for flash writes. An external chip captures the contents of flash memory after each experiment.

To automate the testing of flash write behavior, we use a flash memory testbed designed by Salajegheh [47]. The two major components of the testbed are a test platform and a connected monitoring platform. The monitoring platform is based on an additional MSP430 microcontroller. The test platform runs a test program at low voltage. When the test program completes, the test platform sends the result of the experiment to the monitoring chip via GPIO pins. The test and monitoring platforms share 8+1 GPIO pins to carry one byte of data and a clock signal. Once the test platform puts data on its eight data pins, it raises the clock pin. The monitoring chip reads data from its GPIO pins whenever it detects a rising clock signal and logs the results in its own flash memory. The monitoring chip runs at a voltage above

the nominal minimum for its own flash writes, thereby storing reliably.

3.2.2 Unreliable, Low-Voltage Flash Memory Writes

The TI MSP430 datasheet [24] states that flash writes at any voltage lower than the nominal minimum voltage (which is 2.2V in the case of MSP430F2131) are not guaranteed to succeed. However, as the graphs in Figure 3.2 show, not all flash writes fail at low voltages. On the contrary, in this specific experiment, most of the writes(95.24% at 1.9V and 89.88% at 1.8V) succeed.

In a NOR flash memory, all cells are initialized to 1, and writing data to a byte of flash memory means setting an appropriate number of bits to 0 by applying electrical charge to the corresponding flash cells. At low voltage, there may be insufficient charge to effect a transition to 0, and a flash write may store fewer 0 bits than requested [42]. To be specific, we define errors as follows: when a byte of data d_1 is written in a flash memory address and then data d_2 is read from that address, there is an error if $d_1 \neq d_2$. An experiment, explained next, investigates the behavior of low-voltage flash memory and gives bit-level results.

Using the automated flash testbed explained in Section 3.2.1, the test platform runs a program that writes numbers $\{0, \cdots, 255\}$ to flash memory, then sends the contents of its flash memory to the monitoring platform via GPIO pins. Table 3.2 compares the written data and the intended data for cases in which errors occurred. It demonstrates that, when both are represented as integers, the absolute value of the stored data is always greater than or equal to the absolute value of the intended data.

Table 3.2: Erroneous flash writes at low voltage. Insufficient electrical charge may result in some bits failing to transition from 1 (the initial state) to 0.

(Binary)	Intended	00001100	00001101	00001110	00010100	00100111	10100100
	Written	11101101	01011111	11111111	11111111	00101111	10101111
Hamming distance		4	3	5	6	1	3

3.2.3 Determining Factors That Affect Error Rates

We consider the following potential factors that may affect the error rate of setting a bit to 0 in a flash memory at low voltage: voltage level, Hamming weight of the data, wear-out history, permutation of 0s, and neighbor cells. The effects of each of these variables are evaluated by designing an experiment to test a hypothesis. All the experiments are performed on flash memories with minimal previous usage unless stated otherwise.

Voltage level: Our hypothesis is that the lower a chip's operating voltage (and that of its on-chip flash memory), the higher the error rate of flash writes. Figure 3.3 confirms this hypothesis; moreover, the graph shows that for different chips of exactly the same type, the error rate can be different even under equivalent voltages.

Experiment: Two MSP430F2131 and two MSP430F1232 microcontrollers run a program that writes zeros to the data segment of their flash memory. We increased the microcontroller's operating voltage in 10-mV steps, and used the monitoring platform to compute the byte error rates over 50 runs.

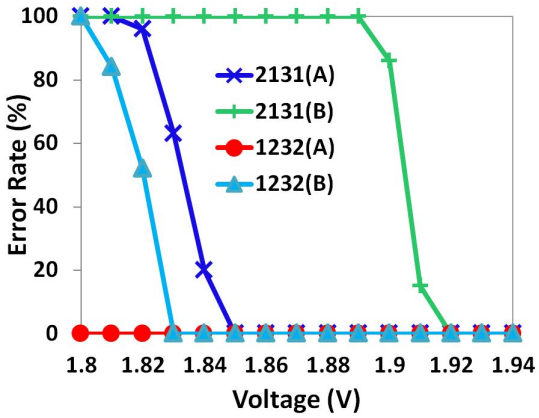


Figure 3.3: Flash write error rates decrease as voltage increases. This trend holds for all the chips (MSP430F2131 and MSP430F1232) we tested, though error rates differ even between chips of the same model.

Hamming weight: In an erased (i.e., having value 1) flash cell, writing a 1 is always error-free because no change to the cell is necessary. However, setting a cell to 0 might fail if there is not enough charge accumulated in that cell. Our hypothesis is that the lower the Hamming weight (number of 1s in the binary representation) of a number, the higher the probability of error when writing that number to flash at low voltage.

Based on per-byte Hamming weight, there are nine equivalence classes of integers that can be represented in one byte. The weight-8 equivalence class has only one member, 255, which can always be written to an erased flash cell without error. The other extreme case is the weight-0 equivalence class, containing only 0s, that requires all eight bits to transition to 0. Figure 3.4 shows the byte error rate for all nine equivalence classes, measured in the following experiment.

Experiment: At 1.84V, a MSP430F2131 runs a program that writes numbers from the same equivalence class to one block (64 bytes) of flash memory. We used the monitoring platform to compute the average byte error rate of flash writes for

each of the nine equivalence classes over 50 runs.

Corollary: To exploit the fact that the Hamming weight of a number affects error rate when written to flash, one can transform numbers into numbers with greater Hamming weights before writing them to flash memory.

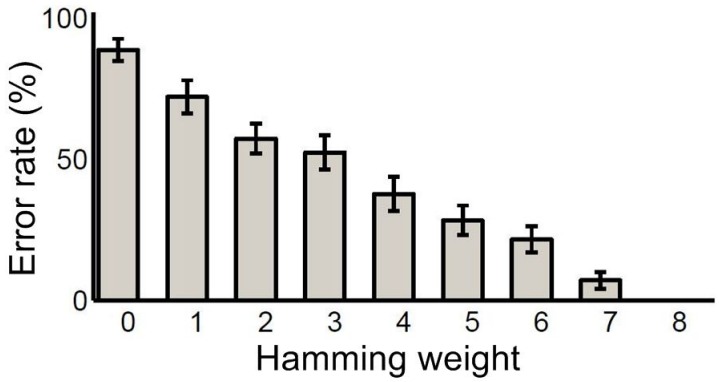


Figure 3.4: As the Hamming weight (number of 1s in the binary representation) of a number increases, the error rate of low-voltage flash write declines. The data corresponds to a MSP430F2131 running at 1.84V.

Wear-out history: Flash memory has a limited lifetime (about 10^5 cycles of erasures) after which the erase operations fail to reset the bits to 1 reliably. We suspect that the more flash memory is erased (worn-out), the lower its error rate of setting bits to 0 would become. This counterintuitive hypothesis is consistent with the notion that flash erasures (settings bits to 1) become harder with wear-out. Figure 3.5 shows a heat map of bit error rate for three blocks of flash memory (192 bytes) on an MSP430F2131 microprocessor. Lighter colors in the heat map represent higher error rates. The disproportionately dark color of the middle block is due to more frequent erasure of that block compared to the other two blocks.

Experiment: A MSP430F2131 runs a program that writes zeros to all three blocks of its flash memory. The MSP430 is first worn out such that one block has 6000

write/erase cycles and two blocks have minimal previous usage. We used the monitoring platform to compute the average error rate for all bits in the three blocks of memory over 50 trials.

Corollary: Wear-out history affects error rate, so storing data in more than one location might decrease the error rate, especially if those locations are in different blocks of memory.

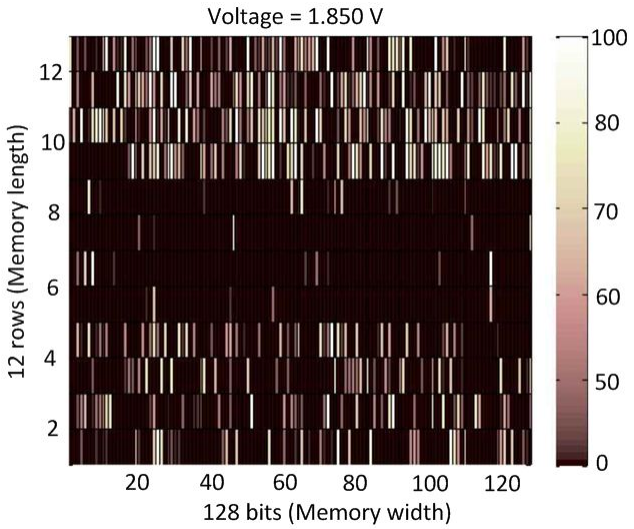


Figure 3.5: Worn-out flash memory blocks are biased toward ease of writing zeros. Lighter color represents higher average number of error over 50 trials. The middle block has been write/ease cycles 6000 times. The other two blocks are minimally used.

Permutation of 0s: Two numbers belonging to the same Hamming weight equivalence class can have different permutations of 0 bits. We tested to see how the error rate depends on the permutation of 0s in one byte of data. For example, the numbers 240, 15, 170, and 71 all have four 0s in their binary representation but in different places (240 has 0s in the right nibble, and 15 has all of its 0s in its left nibble, etc.). The result of the experiment shows a similar byte error rate with mean

of $39.85 \pm 4.29\%$ for numbers in the same equivalence class. The small standard deviation (4.29%) shows that the permutation of 0s does not significantly affect the error rate and therefore we do not consider this to be a factor in our design directions.

Experiment: A MSP430F2131 runs a program that cycles through eight numbers from the same Hamming-weight equivalence class, writing them to 192 consecutive bytes of flash memory. We used the monitoring platform to compute the average error rates for each of the 192 bytes over 50 trials.

Neighbor cells: Another factor that might affect the error rate of storage in a flash cell at low voltage is the values of neighboring cells. However, our results suggest that a cell's error rate does not appear to depend on the values stored in neighboring cells (Figure 3.6).

Experiment: In order to determine if the error rate of a cell is affected by its neighbor, we consider all numbers from the same Hamming-weight equivalence class whose two Least Significant Bits (LSBs) are set to either 00 (case 1) or 10 (case 2). An example of case 1 is number 60 ($0b00111100$) and an example of case 2 is number 30 ($0b00011110$). This experiment fixes the Hamming weight variable and changes the neighbor value of the LSB to be 0 or 1. We deem a write erroneous if the LSB is not set to 0. The experiment was done for a Hamming weight of four and it was repeated for five voltage levels in the interval of 1.82V to 1.84V with steps of 5mV. The error rate for any voltage above 1.84V was close to 0% and for any voltage below 1.82 was close to 100%. We used the monitoring platform to compute the average error rates of case 1 and case 2 for each of the voltage levels over 50 trials.

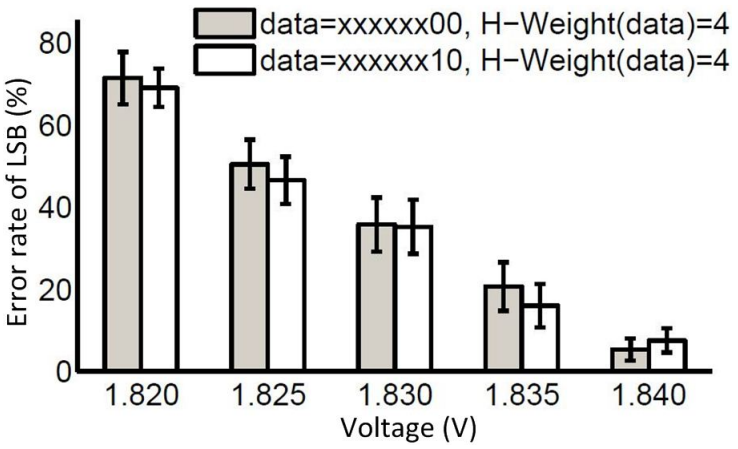


Figure 3.6: Error rate of a cell is not noticeably influenced by the value of its neighbor. The graph shows that the value of the second LSB does not greatly affect the error rate of the LSB. The bars show the error rate of the LSB for writing numbers from the same Hamming-weight equivalence class whose two LSBs are set to either 00 (dark bars) or to 10 (light bars).

3.2.4 Accumulative Memory Behavior

It is helpful to understand a few details of the electrical nature of flash memory in order to appreciate the expected behavior of conventional digital abstractions when layered above embedded flash memory. Each flash memory cell is a floating-gate (FG) transistor made up of a source, drain, control gate, and floating gate. The floating gate is separated from the source and drain by an insulating oxide layer that makes it difficult for electrons to travel into or out of the gate. Flash cells rely on this oxide to maintain logical state in the absence of power, making the memory non-volatile [42].

To write a memory cell (which has an erased value of 1), the control circuitry applies a high field to the source. The application of this field greatly increases the probability that electrons in the floating gate will tunnel to the source. If a sufficient number of electrons tunnel to the source, the cell is subsequently read as a 0. To

erase a cell (that is to restore a 1), the control circuitry applies a high field to both the source and drain. This field energizes the electrons currently stored near the source, allowing them to jump the oxide barrier to the floating gate where they are once again trapped [42].

Not all electrons must transit in order for a write or erase operation to be successful. The operation only needs to change the state of some majority of the electrons so that subsequent read operations detect sufficient charge to discern the intended value. Lowering the applied voltage (and thus the field strength) lowers the probability of state change for each electron but, as noted earlier, electrons that do transit will remain in place.

A low-power storage scheme can benefit from this accumulative property by repeating writes to the same cell. Each write operation will increase the chance of success by forcing some number of state transitions. In other words, a failed write is still progress.

3.3 Design of a Low-Voltage Storage System

This section presents our design for a software system that enables reliable flash memory writes at low voltage. We first present a model that captures the basic characteristics and behavior of flash memory. We then set design goals for the model under consideration. We introduce three methods for reliable flash storage, which we refer to as *in-place writes*, *multiple-place writes*, and *RS-Berger codes*. Each method aims to meet our design goals for reliable non-volatile storage.

3.3.1 Modeling Low-Voltage Flash Memory

A NOR flash memory has a set of n cells that are initially set to 1. We represent the state of the cells by $c_1, \cdots, c_n$; the value of c_i can be 0 or 1. A cell can be set to 0 using a write operation. The $1 \rightarrow 0$ transition might fail at low voltage while

the $1 \rightarrow 1$ will obviously succeed. Flash memory at low voltage, where errors occur only in one direction, can be modeled as a Z-channel. Flash memory is a write-once memory [45] and once a cell is set to 0 (i.e., once it is programmed), it cannot be changed back to 1 without using an erase operation. In flash memory, cells are organized by blocks, and an erase operation resets an entire block of cells. Block erasures are costly in terms of time and energy and they cause wear to flash cells.

Operations: There are two operations in this model: (1) An update operation that changes a subset of cells to 0 to represent a value, and (2) A decoding operation that maps cell states (i.e., memory state) to a value. Updating a variable means changing the values of $c_1, \cdots, c_n$ to $c'_1, \cdots, c'_n$. Assuming that no erase operation occurs, and therefore no bits are reset to 1 after being set to 0, we have $\forall i \in \{1, \cdots, n\}, c_i \geq c'_i$ after an update. If the update operation is performed when operating voltage is below the nominal minimum required for flash memory, the update operation may not be error-free.

3.3.2 Design Goals

Our storage techniques, which aim to provide reliable storage for low-power devices, are designed with the following metrics in mind:

- *Error rate:* The first and foremost design goal is to minimize the error rate to provide applications with reliable non-volatile storage.
- *Energy consumption:* The energy consumed to achieve an acceptably low error rate should not exceed the expected energy savings gained by running at a lower voltage.
- *Delay:* We define delay as the difference between the execution time to store data reliably at a low voltage and to store the same data at a high voltage.

The delay caused by the storage technique should be reasonably small.

3.3.3 Proposed Methods

Toward the design goals discussed previously, we propose methods to deal with errors caused by using flash memory at low voltage.

3.3.3.1 In-Place Writes

Since the transition of a 1 to a 0 in a NOR flash memory at low voltage is stochastic rather than guaranteed, the *in-place writes* method repeats the write of each byte (to the same memory location) more than once if error occurs, up to a *threshold* number of attempts. Algorithm 1 gives the details for Encode and Decode procedures for in-place writes. The *in-place writes* makes an attempt to write a byte into memory, reads that memory address, and if the read result does not match the attempted write value, the algorithm makes another attempt to write that value to the same memory address. The write attempts can be controlled using the *threshold*.

The reason *in-place writes* decrease the error rate is that, as explained in Section 3.2.4, each write attempt in the same memory location increases the accumulated charge and therefore raises the probability of storing the intended bit sequence successfully.

3.3.3.2 Multiple-Place Writes

Another approach to increase the reliability of flash writes at low voltage is to write a value to more than one location in flash memory if error occurs up to a *threshold* number of locations. Later, to retrieve the stored data, the *multiple-place writes* method reads the data from the specified address and several other addresses associated with it, then returns the bitwise AND of all of the stored values. Algorithm 2 details Encode and Decode procedures of the *multiple-place writes* method. The

Algorithm 1 The encoding and decoding algorithms for *in-place writes* method to store *data* to *address* by repeating the writes up to a *threshold* number of attempts if necessary.

Encode($data, address, threshold$)

```
Write_To_Flash(data,address)
result ← Read_From_Flash(address)
repeat ← 1
while (result ≠ data) AND (repeat < threshold) do
    Write_To_Flash(data,address)
    result ← Read_From_Flash(address)
    repeat ← repeat + 1
```

Decode($address$)

```
result ← Read_From_Flash(address)
return result
```

multiple-place writes makes an attempt to write a byte into one memory address, reads that memory address, and if the read result does not match the attempted write value, the algorithm makes another attempt to write that value to a different memory address. The write attempts can be controlled using the *threshold*.

The reason the *multiple-place writes* approach can decrease the error rate is as follows: All cells of flash memory are initially set to 1. An error means that writing a 0 has failed and a bit cell c_i has remained untouched (logical 1) although it was intended to be set to 0. If the cell write in one of the locations has not failed, and cell c_i is 0 in at least one location, getting the AND of the read values from all locations will make cell $c_i = 0$ in the AND result. The case of writing a 1 to a cell does not cause an error since it means changing a cell from 1 to 1.

3.3.3.3 RS-Berger Codes

Our third method to provide reliable flash memory at low voltage involves data coding. We use the concatenation of Reed-Solomon [44]and Berger [6] codes—which we call RS-Berger codes—to detect and correct errors at read time (Algorithm 3).

Algorithm 2 The encoding and decoding algorithms for *muliple-place writes* method to store *data* to *address* by repeating the writes up to *threshold* locations if necessary. The distance between each of these associated locations is $offset$.

Encode$(data, address, threshold, offset)$
  Write_To_Flash(*data*,*address*)
  $result \leftarrow$ Read_From_Flash($address$)
  $repeat \leftarrow 1$
  while $(result \neq data)$ AND $(repeat < threshold)$ **do**
    $phy\_addr \leftarrow address + (repeat \times offset)$
    Write_To_Flash(*data*,*phy_addr*)
    $n\_result \leftarrow$ Read_From_Flash(phy_addr)
    $result \leftarrow result$ & n_result
    $repeat \leftarrow repeat + 1$

Decode$(address, threshold, offset)$
  for $i \leftarrow 0$ **to** $(threshold - 1)$ **do**
    $phy\_addr \leftarrow address + (i \times offset)$
    $n\_result \leftarrow$ Read_From_Flash(phy_addr)
    $result \leftarrow result$ & n_result
  return $result$

Reed-Solomon is a widely used error-correcting code that can correct twice as many erasures as errors. There are three parameters (n, k, d) accompanying the Reed- Solomon (RS) code. The parameter n is the total number of symbols in the codeword, and k is the number of information symbols in the codeword, and the parameter d is the minimum hamming distance of two codewords in the codebook. These three parameters should satisfy the following conditions: $d = n - k + 1$.

A (n, k, d)-RS code can correct up to $\frac{n-k}{2}$ errors and up to $n - k$ erasures. Therefore, if the locations of errors are known, an RS code's error-correcting capacity is improved twofold.

To detect the location of errors and therefore to improve the efficiency of the RS code, we use a Berger code, an error-detecting code that can detect all asymmetric errors [6]. As previously mentioned (Section 3.3.1), flash memory at low voltage can

Algorithm 3 The encoding and decoding algorithms for *RS-Berger codes* writes method. t is the maximum number of erasures RS code can correct.

Encode$(data_{1,\ldots,N}, n)$

```
for i ← 1 to N do
    CW_i ← RS_Encode(data_i,n)
    Write_To_Flash(CW_i,address_i)
for i ← 1 to n do
    for j ← 1 to N do
        sym_{i,j} ← CW_j(i)
    chk_i ← Berger_Encode(sym_{i,(1,..,N)})
    Write_To_Flash(chk_i,address_{N+1} + i − 1)
```

Decode$(addr_{1,\ldots,(N+1)}, n, t)$

```
for i ← 1 to N do
    chk_i ← Read_From_Flash(addr_{N+1} + i − 1)
for i ← 1 to N do
    CW_i ← Read_From_Flash(addr_i)
    for j ← 1 to n do
        sym_{i,j} ← CW_i(j)
errors ← {}
for i ← 1 to n do
    err ← Berger_Decode(sym_{i,(1,..,N)}, chk_i)
    if err = 0 then
        errors ← errors ∪ {i}
if |errors| ≤ t then
    for i ← 1 to N do
        result_i ← RS_Decode(CW_i,errors)
    return result
else
    return "fail to correct errors"
```

be modeled as a Z-channel for which a Berger code is suitable. A Berger codeword consists of two parts: k information bits and $\lceil \log_2(k+1) \rceil$ check bits. The check bits of the Berger codeword represents the number of zeros in the k information bits. Berger code can detect any number of zero-to-one errors, as long as no one-to-zero errors occur in the same codeword. As a particular example, consider the case for

$k = 6$, the information bits are 010010. There are totally 4 zeros in it, therefore, the check bits are 100. One such valid codeword would be 010010 100. When we do the error detection, we check the number of zeros in the information bits part and the binary number in the check bits part. If they are equal, no error occurs, otherwise, errors are detected in the codeword.

We use an $(N + 1) \times n$ matrix to represent RS-Berger codes (Figure 3.7). This matrix has N RS codewords, each of which has n symbols. Each symbol (m bits) is filled in one entry of the matrix. Each column of the matrix, consisting of $m \times N$ bits, supplies the information bits for one Berger code block. After Berger encoding, the $(N + 1)$th row records the check bits for the Berger codes.

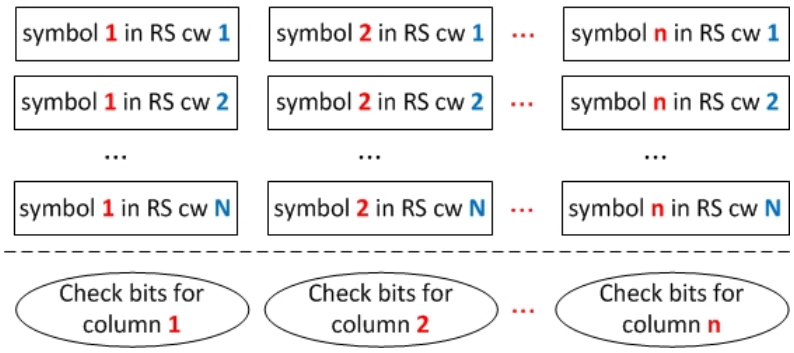


Figure 3.7: Structure of input/output sequence of Berger code.

Figure 3.8 shows how the data are encoded and decoded using our *RS-Berger* code. When encoding the data, we first use RS code to generate n codewords (rows of the matrix) and then we apply a Berger code to compute the check bits for each symbol for all codewords (each column of the matrix). When decoding data, we first use the Berger decoder to check whether or not each column is erroneous. If one entry in the column is erroneous, we consider all the symbols in the column erasures; otherwise, all the symbols in the column are considered correct. Then, once the error

locations are known, we apply RS decoding to correct the erroneous sequences row by row.

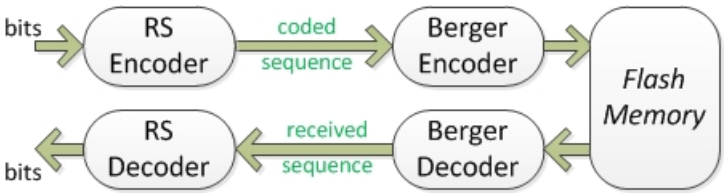


Figure 3.8: A diagram representing the RS-Berger code. An RS-Berger code is the concatenation of the Reed Solomon code and a Berger code.

3.4 Evaluation

Our storage techniques are designed for the resource limitations of low-power devices. In this section, we first evaluate the suitability of the three methods proposed in Section 3.3.3 for low-power devices; we then evaluate the hypothesis that for CPU-bound workloads, operating at low voltage and managing errors is more energy efficient than fixing the operating voltage to the maximum of all the components' nominal minimum voltages.

Summary of results: For a sensor monitoring application that reads 256 data samples from flash memory, aggregates data, and stores the results in flash memory, use of *in-place writes* at 1.8V reduces the energy consumption up to 34% versus running the same application at 2.2V (minimum voltage requirement for the flash memory). This sensing application models a common workload for both wireless sensor nodes and RFID-scale devices.

3.4.1 Comparison of the Proposed Storage Methods

The maximum number of write attempts for both *in-place writes* and *multiple-place writes* methods were set to two. The *RS-Berger* codes used three codewords of size 38 bytes (32 bytes data and 6 bytes parity). These settings enable all three methods to fit their data in 192 bytes of flash memory. Table 3.3 shows the energy consumption and time taken for the same workload under each method. Both in-place writes and multiple-place writes consume less energy and finish more quickly at 1.9V than at 1.8V. Both of these methods are feedback-based and repeat writes if they detect errors. Because there is a lower chance of error at 1.9V, fewer rewrites are required than at 1.8V, so less energy and time are required.

The *in-place writes* method slightly outperforms the *multiple-place writes* method at both voltage levels because its decoding procedure is less CPU-intensive. The *RS-Berger codes* method has the best Error Correction Rate (ECR in Table 3.3) of all. The multiple-place writes method seems to be the most suitable when there are some memory cells that are hard to program and therefore rewriting in those cells is not helpful (Figure 3.5 gives an example of such a case). Compared to *RS-Berger codes* which always guarantee that a certain number of errors can be corrected, the *in-place writes* and *multiple-place writes* methods are less reliable—they offer no such guarantees. Therefore, for applications with a hard reliability requirement, *RS-Berger codes* may be more suitable if the application knows the error rate in advance and is willing to incur extra computational costs for RS-Berger encoding and decoding.

Table 3.3: Performance comparison of the proposed methods at 1.8V and 1.9V. Error Correction Rate (ECR) shows the effectiveness of methods.

Method	**Voltage**	**Time(ms)**	**E(μJ)**	**ECR**
In-place	1.8	24.16	59	96%
M-place	1.8	25.00	63	84%
RS-Berger	1.8	334.45	160	100%
In-place	1.9	15.43	38	100%
M-place	1.9	16.85	40	100%
RS-Berger	1.9	334.73	180	100%

Error Correction Rate: As Table 3.3 illustrates, the two methods that do not use coding—*in-place writes* and *multiple-place writes*—incur similar energy consumption costs. We now compare the effectiveness of these two approaches with respect to the error correction rate.

Figure 3.9 and Figure 3.10 demonstrate that flash storage reliability improves as we increase the number of repeated writes/places at five different voltage levels (all below the nominal minimum voltage for flash writes).

Experiment: Using our automated testbed, the test platform runs a program that writes zeros to 192 consecutive bytes of flash memory (using *in-place writes* and *multiple-place writes* methods in two different experiments). We increase the maximum number of repeated writes from one to ten, one unit at a time. The monitoring platform counts the number of incorrectly stored bytes (those that are not set to zero after the experiment). The experiment was repeated for five different voltages (1.86V-1.90V).

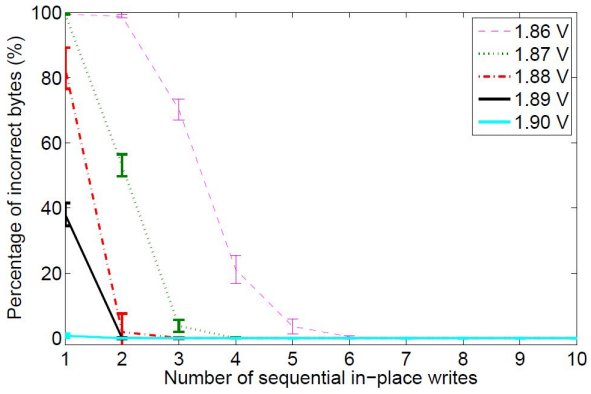


Figure 3.9: Reliability improvement using in-place writes over five voltages.

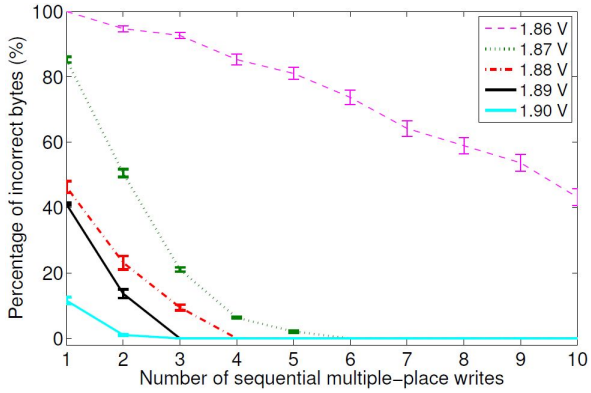


Figure 3.10: Reliability improvement using multiple-place writes over five voltages.

3.4.2 Half-Wits Versus Wits in Practice

To evaluate the end-to-end performance of our storage methods, we have tested a sensor-monitoring application that is CPU-intensive and can benefit from a low-voltage storage. This application reads from flash memory 256 accelerometer samples (each ten bits); computes the maximum, minimum, mean, and standard deviation of the samples; and stores the aggregate information in flash memory. This monitoring application is a blend of CPU and I/O, but it is still a CPU-intensive workload.

Table 3.4 shows that providing the system with a low-voltage storage mechanism via our methods helps to decrease the task's total energy consumption by 34%.

Table 3.4: Energy consumption and execution time for the accelerometer sensor application. At voltage below the recommended (1.8V and 1.9V), in-place writes method with a threshold of two is used.

Method	In-place 1.8V	In-place 1.9V	Standard 2.2V	Standard 3.0V
Clock rate(MHz)	6	6	8	14
Energy(μJ)	270	300	410	760
Time(ms)	151.15	151.32	113.24	64.72

3.4.3 Finding a Crossover Point

We can empirically find the point at which the energy saved on computation compensates for the added cost of repeated flash writes. We compare a workload executed at 2.2V to the same one running at 1.8V using the in-place writes scheme with the threshold k set to 2. We make the worst-case assumption that all data must be written to flash twice (i.e., no bits change on the first attempt). The time spent on flash writes while running at 1.8V is then twice the time spent when operating at 2.2V. We also assume that the clock rate of the system is set to the highest specified for the CPU at each voltage. Specifically, the clock rate would be set to 6 MHz at 1.8V and to 8 MHz at 2.2V.

We empirically determined the power consumption of CPU and flash writes with 1.8V and 2.2V voltage supplies. $P_{C\_1.8} = 1.8mW$, $P_{C\_2.2} = 3.4mW$, $P_{F\_1.8} = 3.7mW$, and $P_{F\_2.2} = 5.8mW$. The variables T_C and T_F are the time spent in computation and on flash memory respectively. With these assumptions, we can write the following

inequality to determine whether a given workload is likely to result in reduced energy consumption:

$$Energy_{1.8} \leq Energy_{2.2} \Rightarrow$$

$$P_{C\_1.8} \times T_{C\_1.8} + P_{F\_1.8} \times k \times T_{F\_1.8} \leq P_{C\_2.2} \times T_{C\_2.2} + P_{F\_2.2} \times T_{F\_2.2} \Rightarrow$$

$$P_{C\_1.8} \times \frac{8MHz}{6MHz} \times T_{C\_2.2} + P_{F\_1.8} \times k \times \frac{8MHz}{6MHz} \times T_{F\_2.2} \leq$$

$$P_{C\_2.2} \times T_{C\_2.2} + P_{F\_2.2} \times T_{F\_2.2}$$

The solution with $k = 2$ is $T_{C\_2.2} \geq 4 \times T_{F\_2.2}$. Therefore, *in-place writes* are competitive over normal flash writes when the time spent on low-voltage operations like computation is at least four times greater than the time spent on flash writes.

3.5 Improvements and Alternatives

This section describes several complementary ways to further improve the performance of our schemes.

3.5.1 Sign Bits and Storing Complements

As discussed in Section 3.2.3, one of the major factors influencing the error rate is the Hamming weight of a number. One way to improve the performance of the low-voltage storage methods is to store numbers with greater Hamming weights ($weight \geq 4$) in flash memory. If a number is lightweight ($weight < 4$), the complement of the number would be stored and a sign bit would be set for future data access. An array of sign bits can be stored separately from the data to avoid disturbing word alignment. A previous work [41] uses a similar technique for multi-level cell (MLC) flash memories with four levels; their techniques result in a significant decrease of energy consumption. The *sign-bit* approach involves very lightweight computation (counting the number of ones) and increases the number of writes by a factor of

one-eighth. Therefore, the effect of this improvement on energy consumption and delay should be comparatively small.

3.5.2 Memory Mapping Table

To exploit the fact that numbers with greater Hamming weights have a lower probability of error, we can also map the most frequently used numbers in the user's data to the heavier numbers. The solution we suggest is to preprocess the data to sort numbers based on their frequency of use. A simple memory mapping table would map the most frequent numbers to the heaviest numbers. Such a table could be preloaded in flash memory so that storing the table would not consume energy at run time. Use of a memory mapping table would only increase the number of reads and would not increase the number of writes. Therefore, the energy consumption overhead and the delay should be smaller than the *sign bit* method.

4. CONTENT-ASSISTED FILE DECODING FOR NON-VOLATILE MEMORIES

Non-volatile memories (NVMs) such as flash memories play a significant role in meeting the data storage requirements of today's computation activities. The rapid increase of storage density for NVMs however brings reliability issues due to closer alignment of adjacent cells on chip, and more levels that are programmed into a cell. We propose a new method for error/erasure correction, which uses the random access capability of NVMs and the redundancy that inherently exists in information content. Although it is theoretically possible to remove the redundancy via data compression, existing coding algorithms do not remove all of it for efficient computation. The method named *content-assisted decoding* can be combined with existing storage solutions for text files. Using the statistical properties of words and phrases in the text of a given language, our decoder identifies the location of each subcodeword representing some word in a given input noisy codeword, and decode receiving bits sequence to compute a most likely word sequence. In this work, we focus on the erasures recovery. The decoder can be adapted to work together with traditional error-correcting codes decoders to keep the number of errors after erasure recovery within the correction capability of traditional ECC decoders. The combined decoding framework is evaluated with a set of benchmark files.

4.1 Introduction

Non-volatile memories, especially flash memories featuring excellent I/O speed and decent storage capacity have attracted great attention from the data storage community. Flash memories are considered one of the most promising candidates for replacing mechanical hard disks in the near future [42]. Towards this goal, significant

progress has been made for increasing the storage density and the endurance of flash memories.

However, higher storage density brings important reliability challenges [23]. The existing solution for reliable storage are usually solely making use of error-correcting codes. In order to satisfy the high data reliability requirement (the error rate should be no more than 10^{-20} after decoding), the error-correcting codes need many more parity check bits to reach the error correction capacity, which is inconsistent with the high density storage idea. In this chapter, we propose a new method for erasure correction named *content-assisted decoding*. Our method uses the fast random access capability of non-volatile memories and the redundancy that inherently exists in information content. By looking up the dictionaries storing the statistical properties of words and phrases of the same language, our decoder first finds the "space" symbol's locations, then breaks the input noisy codeword into subcodewords, with each subcodeword corresponding to a set of possible words. The decoder then recovers the erasures in each noisy subcodeword to select a most likely word sequence as the correction. The new scheme can be combined with existing storage solutions for text files and improve the system's erasure correction capacity. Consider the example in Figure 4.1.

	Codeword	**Text**
Huffman encoding	$(1,0,0,0,0,1,1,1)$	*I am*
LDPC encoding	$(1,0,0,0,0,1,1,1,\mathbf{0},\mathbf{1},\mathbf{1},\mathbf{0})$	*I am*
Noise received	$(1,e,0,e,0,e,1,e,\mathbf{0},\mathbf{e},\mathbf{1},\mathbf{e})$	$\times$
LDPC decoding failure	$(1,e,0,e,0,e,1,e,\mathbf{0},\mathbf{e},\mathbf{1},\mathbf{0})$	$\times$
Content-assisted decoding	$(1,0,0,0,0,1,1,1,\mathbf{0},\mathbf{e},\mathbf{1},\mathbf{0})$	*I am*
LDPC decoding success	$(1,0,0,0,0,1,1,1,\mathbf{0},\mathbf{1},\mathbf{1},\mathbf{0})$	*I am*

Figure 4.1: An example on correcting erasures in the codeword of a text.

The English text *"I am"* is stored using the Huffman coding: $\{I \rightarrow (1,0), \sqcup \rightarrow (0,0), a \rightarrow (0,1), m \rightarrow (1,1)\}$, where $\sqcup$ denotes a space. The information bits are encoded with a Low Density Parity Check (LDPC) code with parity check matrix H (the bold bits denote the parity check bits).

$$H = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Assume that six erasures (marked by the symbol "e") are received by the codeword. The number of erasures exceeds the code's correction capability, and LDPC decoding fails. Our decoder takes in the noisy codeword, and corrects the erasures in the information symbols by looking up a dictionary which contains two words $\{I, am\}$. This brings the number of erasures down to one. Therefore, the second trial of LDPC decoding succeeds, and all the erasures are corrected. Our approach is suitable for natural languages, and can potentially be extended to other types of data where the redundancy in information content is not fully removed by data compression. The dictionaries are preloaded in the flash memory. The scheme takes advantage of the fast random access speed provided by flash memories for fast dictionary look-up and content verification so that the dictionary look-up process in our decoding algorithm could be linear time. For performance evaluation, we have tested a decoding framework that combines a modified soft decision decoder of LDPC codes and our scheme with a set of text file benchmarks. Experimental results show that our decoder indeed increases the correction capability of the LDPC decoder and recovers the erasures efficiently.

The rest of the chapter is organized as follows. Section 4.2 presents the prelimi-

naries, and defines the text file decoding problem. Section 4.3 specifies the algorithms of the content-assisted file decoder. Section 4.4 discusses implementation details and experimental results.

4.2 The Models of File Decoding

In this section, we first introduce the terminologies used in this chapter, then we describe the model of the data storage channel and define the file decoding problem.

4.2.1 Notations

Let $\mathbf{x}$ denote a binary *codeword* $(x_1, x_2, \cdots, x_n) \in \{0,1\}^n$, and we use $\mathbf{x}[i:j]$ to represent the *subcodeword* $(x_i, x_{i+1}, \cdots, x_j)$ and $\mathbf{x}[i]$ to represent the *subcodeword* $(x_1, x_2, \cdots, x_i)$ for short. Let $\mathbf{x}'$ denote a trinary *noisy codeword* $(x'_1, x'_2, \cdots, x'_n) \in \{0,1,e\}^n$. Let the function length($\mathbf{x}$) compute the length of a codeword $\mathbf{x}$ and n_erasure($\mathbf{x}'$) compute the number of erasures in a noisy codeword $\mathbf{x}'$. We say $\mathbf{x}$ to be a *solution* of $\mathbf{x}'$, if for $1 \leq i \leq$ length($\mathbf{x}$),

$$\begin{cases} x_i = x'_i & \text{when } x'_i \neq e \\ x_i = 0 \text{ or } 1 & \text{when } x'_i = e \end{cases}$$

Let $\mathcal{A}$ be an alphabet set, and let $s \in \mathcal{A}$ be a symbol. We denote a *space* by $\sqcup \in \mathcal{A}$. A *word*

$$\mathbf{w} \triangleq (s_1, s_2, \cdots, s_n)$$

of length n is a finite sequence of symbols without any space. A *phrase*

$$\mathbf{p} \triangleq (\mathbf{w}_1, \sqcup, \mathbf{w}_2)$$

is defined as a combination of two words separated by a space. Define a text

$$\mathbf{t} \triangleq (\mathbf{w}_1, \sqcup, \mathbf{w}_2, \sqcup, \cdots, \sqcup, \mathbf{w}_n)$$

as a sequence of words separated by $\sqcup$. A *word dictionary*

$$D_w \triangleq \{[\mathbf{w}_1 : p_1], [\mathbf{w}_2 : p_2], \cdots\}$$

is a finite set of records where a record $[\mathbf{w} : p]$ has a key $\mathbf{w}$ and a value $p > 0$. The value p is an average probability that the word $\mathbf{w}$ occurs in a text. Similarly, a *phrase dictionary*

$$D_p \triangleq \{[\mathbf{p}_1 : p_1], [\mathbf{p}_2 : p_2], \cdots\}$$

stores the probabilities that a set of phrases appear in any given text. In our scheme, it refers to the set of valid phrases ("word combinations") used in files.

The dictionary look-up operations denoted by $D_w[\mathbf{w}]$ and $D_p[\mathbf{p}]$ return the probabilities of words and phrases, respectively. We use the notation $\mathbf{w} \triangleright D_w$ (or $\mathbf{p} \triangleright D_p$) to indicate that there is a record in D_w (or D_p) with key $\mathbf{w}$ (or $\mathbf{p}$). Let π_s be a bijective mapping from a symbol to a binary codeword, and let $\mathbf{x}_s = \pi_s(\sqcup)$. In this work, the mapping π_s is used during data compression before ECC encoding, and it encodes each symbol separately. In the example of Section 4.1, π_s refers to the Huffman codes.

$$\{I \to (1,0), \sqcup \to (0,0), a \to (0,1), m \to (1,1)\}$$

The bijective mapping from a word $\mathbf{w} = (s_1, \cdots, s_n)$ to its binary codeword is defined

as

$$\pi_w(\mathbf{w}) \triangleq (\pi_s(s_1), \cdots, \pi_s(s_n))$$

and the bijective mapping from a text to its binary representation is defined as

$$\pi_t(\mathbf{t}) \triangleq (\pi_w(\mathbf{w}_1), \mathbf{x}_s, \cdots, \mathbf{x}_s, \pi_w(\mathbf{w}_n))$$

where $\mathbf{x}_s = \pi_s(\sqcup)$. We use π_s^{-1}, π_w^{-1} and π_t^{-1} to denote the corresponding inverse mappings.

4.2.2 File Decoding Model

The model of the data storage channel is shown in Figure 4.2.

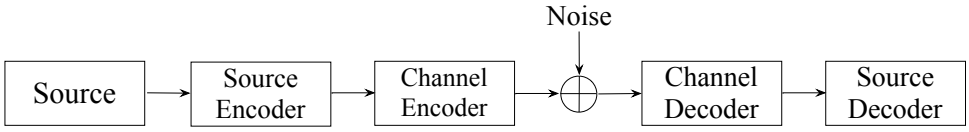


Figure 4.2: The channel model for data storage.

A text $\mathbf{t}$ is generated by the source. The text is compressed by the source encoder (e.g. Huffman encoder), producing a binary codeword $\mathbf{y} = \pi_t(\mathbf{t}) \in \{0,1\}^k$. The compressed bits are fed to a channel encoder (e.g. LDPC encoder), obtaining an ECC codeword $\mathbf{x} = \psi(\mathbf{y}) \in \{0,1\}^n$ where $n > k$. Here we assume a systematic ECC is used. The codeword is then stored by memory cells, and receives some erasures. In this work, a binary erasure channel (BEC) with bit-erasure rate f is assumed. When the cells are read, the channel outputs a noisy codeword $\mathbf{x}' = (x'_1, x'_2, \cdots, x'_n)$ where $x'_i = x_i$ or e, and $1 \leq i \leq n$. The noisy codeword is first corrected by a

channel decoder (e.g. our proposed decoder), producing an estimated ECC codeword $\hat{\mathbf{y}} = \psi^{-1}(\mathbf{x}')$. The source decoder decompresses the corrected codeword, and returns an estimated text $\hat{\mathbf{t}} = \pi_t^{-1}(\hat{\mathbf{y}})$ upon success.

This work focuses on designing better channel decoders ψ^{-1} for correcting bit erasures in text files. We propose a new decoding framework which connects a traditional ECC decoder with a *content-assisted decoder* (CAD) as shown in Figure 4.3.

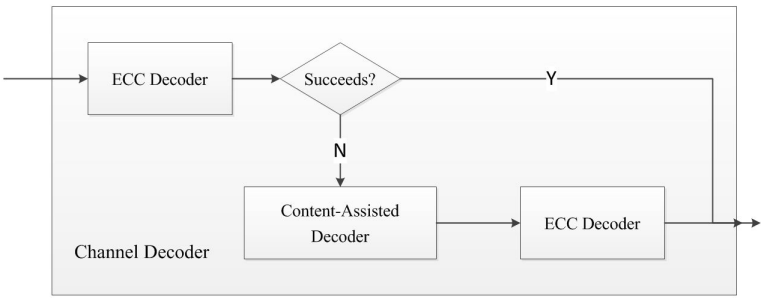


Figure 4.3: The work-flow of a channel decoder with content-assisted decoding.

A noisy codeword is first passed into an ECC decoder. If decoding fails, the decoding output is passed to CAD. With the statistical information stored in D_w and D_p, the CAD selects a word for each subcodeword to form a likely text as the correction for the noisy codeword and reduces the nubmer of erasures left in the codeword. The corrected text is fed back to the ECC decoder to further recover the text. The text file decoding problem for our CAD is defined as follows.

Definition 9. *Let* $\mathbf{t}$ *be some text generated from the source, and let* $\mathbf{x}' \in \{0, 1, e\}^n$ *be a noisy channel output codeword of* $\mathbf{t}$*. Given two dictionaries* D_w *and* D_p*, the text file decoding problem for the CAD is to find an estimated text* $\hat{\mathbf{t}}$ *which is the most*

likely correction for $\mathbf{x}'$, i.e.

$$\underset{\hat{\mathbf{t}}}{\operatorname{argmax}} \Pr\{\hat{\mathbf{t}} \mid \mathbf{x}', D_p, D_w\}.$$

4.3 The Content-Assisted Decoding Algorithms

The content-assisted decoder approximates the solution to the optimization problem in Definition 9 in the three steps: (1) estimate "space" positions in the noisy codeword to divide the codeword into subcodewords, with each subcodeword representing a set of candidate words. (2) Resolve ambiguity by selecting a word for each subcodeword to form a most likely sequence. (3) Connecting the results of step (2) as the input to a modified LDPC soft decision decoder to further reduce error rate. We describe the algorithms of each step in this section.

4.3.1 Creating Dictionaries

The dictionaries D_w and D_p are used in our decoding algorithms. To create the dictionaries, we simply count the frequencies of words and phrases of two words which appear in a relatively large set of different texts in the same language as the texts generated by the source. Fast dictionary look-up is achieved by storing the dictionaries in a content-addressable way thanks to the random access in flash memories, *i.e.*, the probability in a dictionary record is addressed by the value of the corresponding word or phrase. As we show later in section 4.4, the completeness of the dictionaries effects the decoding performance.

4.3.2 Codeword Segmentation

We are aiming to assign 0 or 1 to those erasure bits so that we can recover the noisy codeword to a sequence of words separated by "spaces". Considering the fact that the dictionary is complete or near complete, the probability that the input

text $\mathbf{t}$ contains a word $\mathbf{w}$ that is not in the dictionary ($\mathbf{w} \not\triangleright D_w$) is very low. The erasures are evenly distributed in the information bits. Therefore, after decoding, the erasures located in the words that are not in the dictionary should be as few as possible. If the dictionary is complete, the number of erasures located in the non-dictionary words is 0 because all words are in the dictionary. By this intuition, we define the codeword segmentation function σ in the following way: σ takes in a noisy codeword $\mathbf{x}$ and a word dictionary D_w, then assign 0 or 1 to erasure bits to make the corrected codeword represent a text, *i.e.*, a sequence of valid words separated by spaces, and the number of erasures located in non-dictionary words is minimized. If $\sigma(\mathbf{x}, D_w) = ((\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_k), (i_1, i_2, \cdots, i_{k-1}))$, where the number of records $|D_w|$ is bounded by some constant K, and $i_j \in \mathbb{N}$ is the index of the first bit of the j-th space in $\mathbf{x}$, the subcodeword $\mathbf{x}_1 = \mathbf{x}[i_1 - 1]$, $\mathbf{x}_k = \mathbf{x}[i_{k-1} + \text{length}(\mathbf{x}_s) : \text{length}(\mathbf{x})]$, and $\mathbf{x}_j = \mathbf{x}[i_{j-1} + \text{length}(\mathbf{x}_s) : i_j - 1]$ for $j \in \{2, 3, \cdots, k-1\}$. The mapping σ is required to satisfy the following properties:

1. for each subcodeword $\mathbf{x}_j$, $1 \leq j \leq k$, $\exists \mathbf{w}$ such that $\pi_w(\mathbf{w})$ is a *solution* to $\mathbf{x}_j$.

2. $\sum_{j=1}^{k} \text{n\_erasure}(\mathbf{x}_j) \times \text{flag}(j)$ is minimized, where

$$\text{flag}(j) = \begin{cases} 0 & \text{if } \exists \mathbf{w} \triangleright D_w \text{ such that } \pi_w(\mathbf{w}) \text{ is a } \textit{solution} \text{ to } \mathbf{x}_j \\ 1 & \text{otherwise} \end{cases}$$

Let the cost function $\text{c}(i)$ return the minimum number of erasures located in the non-dictionary words after converting the subcodeword $\mathbf{x}[i]$ to represent a text. We have the following recurrence for $i \geq lmin$:

$$\text{c}(i) \triangleq \min\{ \min_{k=lmin}^{\min\{i, lmax\}} \{\text{c}(i - l_s - k) + \text{g}(i - k - l_s + 1, i - k) + \text{h}(i - k + 1, i)\}, \text{h}(1, i)\}$$

where $lmin/lmax$ are the shortest/longest codeword length for a single dictionary word respectively, $l_s = \text{length}(\mathbf{x}_s)$, and when $i < lmin$, $c(i) = \infty$. Clearly, if the dictionary are complete, $\text{c}(n) = 0$, where $n = \text{length}(\mathbf{x})$.

The function $\text{g}(i, j)$ denotes whether $\mathbf{x}[i : j]$ can be decoded to $\sqcup$. The function $\text{h}(i, j)$ computes the cost taken to obtain a single word of subcodeword length $j - i$.

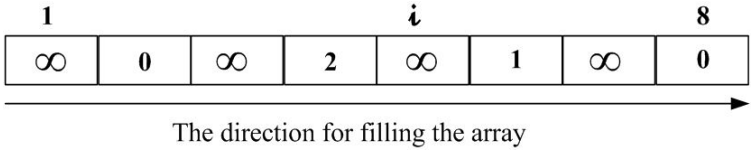
$$\text{g}(i,j) \triangleq \begin{cases} 0 & \text{if } \mathbf{x}_s \text{ is a } \textit{solution} \text{ to } \mathbf{x}[i:j] \\ \infty & \text{otherwise} \end{cases}$$

$$\text{h}(i,j) \triangleq \begin{cases} 0 & \text{if } \exists \mathbf{w} \triangleright D_w, \pi_w(\mathbf{w}) \text{ is a } \textit{solution} \text{ to } \mathbf{x}[i:j] \\ \text{n\_erasure}(\mathbf{x}[i:j]) & \text{else if } \exists \mathbf{w} \not\triangleright D_w, \pi_w(\mathbf{w}) \text{ is a } \textit{solution} \text{ to } \mathbf{x}[i:j] \\ \infty & \text{otherwise} \end{cases}$$

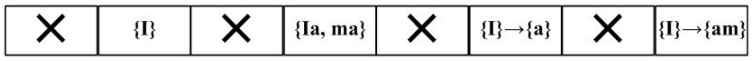
Example 10. *Consider the example in section 4.1. The input noisy codeword* $\mathbf{x}' = (1, e, 0, e, 0, e, 1, e)$*, and the word dictionary* $D_w = \{[I : 0.5], [am : 0.5]\}$*. We have* $lmin = \text{length}(\mathbf{x}_s) = 2$*,* $lmax = 4$ *and* $\sigma(\mathbf{x}', D_w) = (((1, e), (0, e, 1, e)), (3))$*. Starting from* $c(1)$*, we recursively compute* $\text{c}(i)$ *for all* $1 \leq i \leq n$ *and* n *is the codeword length. The results are shown in Figure 4.4a. For instance, to compute* $\text{c}(2)$*, we first compute* $\text{c}(1) = \infty$ *because the codeword length is at least* 2*. We then compute* $\text{h}(1, 2) = 0$*. This is because we can assign* 0 *to the first erasure and make* $\mathbf{x}_1 = (10)$*, which can be decoded as "I". Finally, we have* $\text{c}(2) = \min(0, \infty) = 0$*.*

Our objective is to compute $\text{c}(n)$ given an input codeword of length n, and find out the space positions which helps achieve the minimum cost. When $\text{c}(i)$ is computed recursively starting from $\text{c}(1)$, some entries will be called several times. For instance, in Example 10, the entry $\text{c}(2)$ needs to be computed when we compute $\text{c}(i)$

for $i > 2$. A good way for speeding up such computation is to use dynamic programming techniques shown in Algorithm 4, which computes the final result iteratively starting from c(1), an entry computed in the previous iteration is saved for later iterations.



(a) Array c



(b) Array m

Figure 4.4: The examples of codeword segmentation. In Figure (b): sets of words means the subcodeword $\mathbf{x}[i]$ can be decoded to a word or word sequence chosen from any word in the word set. The $\rightarrow$ defines the word sequence order. The cross $\times$ represents a subcodeword $\mathbf{x}[i]$ can neither be decoded to a word nor to a word sequence.

The algorithm treats c(i) as the entries of one dimensional array. Starting from c(1), the algorithm fills each entry from $c(1)$ to $c(n)$, as shown in Figure 4.4a. The corresponding space locations for breaking the subcodeword $\mathbf{x}[i]$, and the set of word sequences that $\mathbf{x}[i]$ can be decoded to represent are recorded using a one dimensional array m. In practice, as f is close to 0, the average number of erasures in the subcodeword $\mathbf{x}[k:j]$ is small. The cardinality of the set of possible words S_w for a given noisy subcodeword $\mathbf{x}[k:j]$ can be bounded by $2^{\text{n\_erasure}(\mathbf{x}[k:j])}$. In practice, we first brute force search the set $\{\mathbf{w} \mid \mathbf{w} \triangleright D_w, \pi_w(\mathbf{w})$ is a *solution* to $\mathbf{x}[k:j]\}$ and record the results in m. If $\mathbf{x}[k:j]$ cannot be decoded to any word in the dictionary, the set for non-dictionary $\{\mathbf{w} \mid \mathbf{w} \not\triangleright D_w, \pi_w(\mathbf{w})$ is a *solution* to $\mathbf{x}[k:j]\}$ is computed

and recorded in m. As we are interested in the space locations for the whole input noisy codeword, after the entries of c and m have been filled (Figure 4.4), we get the optimal solution from $m(n)$. The results are the ordered space locations and the sets of words for the subcodewords between the spaces. Assume that the subcodeword of each word has limited length bounded by some constant and the number of erasures in each word is small, the time complexity of our dynamic programming algorithm is $\mathcal{O}(n^2)$, and $\mathcal{O}(n)$ space is used for storing the arrays c and m.

Algorithm 4 CodewordSegmentation($\mathbf{x}, D_w, lmin, lmax$)

$n \leftarrow$ length($\mathbf{x}$), $l \leftarrow$ length($\mathbf{x}_s$)
Let c be an array of length n
Let *wordSets* and *spaces* be two arrays of empty lists
for i from 1 to n **do**
  $c(i) = \infty$
for i from $lmin$ to n **do**
  $flag = 0$,$k = lmin$
  while $flag \neq 0$ AND $k \leq \min(lmax, i)$ **do**
    if $k = i$ **then**
      Brute force assign 0 or 1 to erasures in $\mathbf{x}[k]$
      $S_w \leftarrow \{\mathbf{w} \mid \mathbf{w} \triangleright D_w, \pi_w[\mathbf{w}]$ is a *solution* to $\mathbf{x}[k]\}$
      if $S_w \neq \emptyset$ **then**
        $c(k) = 0$, $wordSets(k) = S_w$
      else
        $S_w \leftarrow \{\mathbf{w} \mid \mathbf{w} \not\triangleright D_w, \pi_w[\mathbf{w}]$ is a *solution* to $\mathbf{x}[k]\}$
        if $S_w \neq \emptyset$ AND $c(i) >$ n_erasure($\mathbf{x}[k]$) **then**
          $c(i) =$ n_erasure($\mathbf{x}[k]$), $wordSets(k) = S_w$
    else
      if $c(i-k-l) \neq \infty$ AND $\mathbf{x}_s$ is a *solution* to $\mathbf{x}[i-k-l+1 : i-k]$ **then**
        Brute force assign 0 or 1 to erasures in $\mathbf{x}[i-k+1 : i]$
        $S_w \leftarrow \{\mathbf{w} \mid \mathbf{w} \triangleright D_w, \pi_w[\mathbf{w}]$ is a *solution* to $\mathbf{x}[i-k+1 : i]\}$
        if $S_w \neq \emptyset$ AND $c(i) > c(i-k-l)$ **then**
          $c(i) = c(i-k-1)$, $wordSets(i) = wordSets(i-k-l) \rightarrow S_w$
          $spaces(i) = spaces(i-k-l) \rightarrow i-k-l+1$
        else
          $S_w \leftarrow \{\mathbf{w} \mid \mathbf{w} \not\triangleright D_w, \pi_w[\mathbf{w}]$ is a *solution* to $\mathbf{x}[i-k+1 : i]\}$
          if $S_w \neq \emptyset$ AND $c(i) > c(i-k-l) +$ n_erasure($\mathbf{x}[i-k+1 : i]$) **then**
            $c(i) = c(i-k-l) +$ n_erasure($\mathbf{x}[i-k+1 : i]$), $wordSets(i) = wordSets(i-k-l) \rightarrow S_w$
            $spaces(i) = spaces(i-k-l) \rightarrow i-k-l+1$
    if $c(i) = 0$ **then**
      $flag = 1$
    $k++$
return $wordSets(n)$ and $spaces(n)$

Example 11. *For the example in section 4.1, the tables c and m computed by Algorithm 4 are shown in Figure 4.4. The minimum decoding cost is* $c(8) = 0$*, which means the noisy codeword can be decoded as a sequence of dictionary words. And the index of the estimated space is* 3*. With the estimated space, the subcodeword* $\mathbf{x}[1:2] = (1, e)$ *can be decoded to a word in the set* $\{I\}$*, and the subcodeword* $\mathbf{x}[5:8] = (0, e, 0, e)$ *can be decoded a word in the set* $\{am\}$*.*

4.3.3 Ambiguity Resolution

Given the subcodewords $(\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_k)$ between the estimated spaces, and a list of word sets $(\mathbf{W}_1, \mathbf{W}_2, \cdots, \mathbf{W}_k)$ computed from the codeword segmentation algorithm, for $i \in \{1, \cdots, k\}$ we select a word $\mathbf{w}_i$ from $\mathbf{W}_i$ to form a most probable text $\hat{\mathbf{t}} = (\mathbf{w}_1, \sqcup, \mathbf{w}_2, \sqcup, \cdots, \sqcup, \mathbf{w}_k)$. The codeword $\pi_t(\hat{\mathbf{t}})$ is a correction for the input noisy codeword. Specifically, this step is to compute

$$\begin{aligned} & \operatorname{argmax}_{(\mathbf{w}_1, \mathbf{w}_2, \cdots, \mathbf{w}_k) \in \mathbf{W}_1 \times \mathbf{W}_2 \cdots \times \mathbf{W}_k} \Pr\{(\mathbf{w}_1, \mathbf{w}_2, \cdots, \mathbf{w}_k) \mid (\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_k)\} \\ = \; & \operatorname{argmax}_{(\mathbf{w}_1, \mathbf{w}_2, \cdots, \mathbf{w}_k) \in \mathbf{W}_1 \times \mathbf{W}_2 \cdots \times \mathbf{W}_k} \Pr\{(\mathbf{w}_1, \mathbf{w}_2, \cdots, \mathbf{w}_k), (\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_k)\} \end{aligned}$$

Let the function $\mathrm{P}(\mathbf{w}_i)$ compute the maximal joint probability when some word $\mathbf{w}_i$ is selected from $\mathbf{W}_i$ and appended to the previously selected word sequence $(\mathbf{w}_1, \mathbf{w}_2, \cdots, \mathbf{w}_{i-1})$. For $i \in [2, k]$, we have

$$\begin{aligned} \mathrm{P}(\mathbf{w}_i) \; & \triangleq \max_{(\mathbf{w}_1, \cdots, \mathbf{w}_{i-1}) \in \mathbf{W}_1 \times \cdots \times \mathbf{W}_{i-1}} \; \Pr\{(\mathbf{w}_1, \cdots, \mathbf{w}_i), (\mathbf{x}_1, \cdots, \mathbf{x}_i)\} \\ & = \max_{(\mathbf{w}_1, \cdots, \mathbf{w}_{i-1}) \in \mathbf{W}_1 \times \cdots \times \mathbf{W}_{i-1}} \; \Pr\{\mathbf{w}_1\} \Pr\{\mathbf{x}_1 \mid \mathbf{w}_1\} \Pr\{\mathbf{w}_2 \mid \mathbf{w}_1\} \Pr\{\mathbf{x}_2 \mid \mathbf{w}_2\} \\ & \qquad \Pr\{\mathbf{w}_3 \mid (\mathbf{w}_1, \mathbf{w}_2)\} \Pr\{\mathbf{x}_3 \mid \mathbf{w}_3\} \cdots \\ & \qquad \Pr\{\mathbf{w}_i \mid (\mathbf{w}_1, \mathbf{w}_2, \cdots, \mathbf{w}_{i-1})\} \Pr\{\mathbf{x}_i \mid \mathbf{w}_i\} \end{aligned}$$

Assume the words in a text form a one-step Markov chain, *i.e.*, for $i \geq 2$,

$$\Pr\{\mathbf{w}_i \mid (\mathbf{w}_1, \mathbf{w}_2, \cdots, \mathbf{w}_{i-1})\} = \Pr\{\mathbf{w}_i \mid \mathbf{w}_{i-1}\}$$

Therefore, we rewrite the equation above as:

$$\begin{aligned}
\mathrm{P}(\mathbf{w}_i) &= \max_{(\mathbf{w}_1,\cdots,\mathbf{w}_{i-1})\in \mathbf{W}_1\times\cdots\times\mathbf{W}_{i-1}} \\
&\quad \Pr\{\mathbf{w}_1\}\Pr\{\mathbf{x}_1 \mid \mathbf{w}_1\}\Pr\{\mathbf{w}_2 \mid \mathbf{w}_1\}\Pr\{\mathbf{x}_2 \mid \mathbf{w}_2\} \\
&\quad \Pr\{\mathbf{w}_3 \mid \mathbf{w}_2\}\Pr\{\mathbf{x}_3 \mid \mathbf{w}_3\}\cdots\Pr\{\mathbf{w}_i \mid \mathbf{w}_{i-1}\}\Pr\{\mathbf{x}_i \mid \mathbf{w}_i\} \\
&= \max_{(\mathbf{w}_1,\cdots,\mathbf{w}_{i-1})\in \mathbf{W}_1\times\cdots\times\mathbf{W}_{i-1}} \\
&\quad \Pr\{\mathbf{w}_1\}\Pr\{\mathbf{w}_2 \mid \mathbf{w}_1\}\cdots\Pr\{\mathbf{w}_i \mid \mathbf{w}_{i-1}\}\prod_{k=1}^{i}\Pr\{\mathbf{x}_k \mid \mathbf{w}_k\} \\
&= \max_{(\mathbf{w}_1,\cdots,\mathbf{w}_{i-1})\in \mathbf{W}_1\times\cdots\times\mathbf{W}_{i-1}} \Pr\{\mathbf{w}_i \mid \mathbf{w}_{i-1}\}\Pr\{\mathbf{x}_i \mid \mathbf{w}_i\} \\
&\quad \Pr\{\mathbf{w}_1\}\Pr\{\mathbf{w}_2 \mid \mathbf{w}_1\}\cdots\Pr\{\mathbf{w}_{i-1} \mid \mathbf{w}_{i-2}\}\prod_{k=1}^{i-1}\Pr\{\mathbf{x}_k \mid \mathbf{w}_k\} \\
&= \max_{(\mathbf{w}_1,\cdots,\mathbf{w}_{i-1})\in \mathbf{W}_1\times\cdots\times\mathbf{W}_{i-1}} \Pr\{\mathbf{w}_i \mid \mathbf{w}_{i-1}\}\Pr\{\mathbf{x}_i \mid \mathbf{w}_i\} \\
&\quad \Pr\{(\mathbf{w}_1, \cdots, \mathbf{w}_{i-1}), (\mathbf{x}_1, \cdots, \mathbf{x}_{i-1})\} \\
&= \max_{\mathbf{w}_{i-1}\in \mathbf{W}_{i-1}} \Pr\{\mathbf{x}_i \mid \mathbf{w}_i\}\Pr\{\mathbf{w}_i|\mathbf{w}_{i-1}\} \\
&\quad \max_{(\mathbf{w}_1,\cdots,\mathbf{w}_{i-2})\in \mathbf{W}_1\times\cdots\times\mathbf{W}_{i-2}} \Pr\{(\mathbf{w}_1, \cdots, \mathbf{w}_{i-1}), (\mathbf{x}_1, \cdots, \mathbf{x}_{i-1})\} \\
&= \max_{\mathbf{w}_{i-1}\in \mathbf{W}_{i-1}} \Pr\{\mathbf{x}_i \mid \mathbf{w}_i\}\Pr\{\mathbf{w}_i|\mathbf{w}_{i-1}\}\,\mathrm{P}(\mathbf{w}_{i-1})
\end{aligned}$$

and

$$\mathrm{P}(\mathbf{w}_1) = \Pr\{\mathbf{w}_1\}\Pr\{\mathbf{x}_1 \mid \mathbf{w}_1\}$$

The conditional probability $\Pr\{\mathbf{x}_i \mid \mathbf{w}_i\}$ is computed from the channel statistics by

$$\Pr\{\mathbf{x}_i \mid \mathbf{w}_i\} = f^{\mathrm{n\_erasure}(\mathbf{x}_i)}(1-f)^{\mathrm{length}(\mathbf{x}_i)-\mathrm{n\_erasure}(\mathbf{x}_i)}$$

Since the number of erasures in $\mathbf{x}_i$ is fixed, $\Pr\{\mathbf{x}_i \mid \mathbf{w}_i\}$ is the same for all $\mathbf{w}_i \in \mathbf{W}_i$. We are aiming to find a word sequence $(\mathbf{w}_1, \mathbf{w}_2, \cdots, \mathbf{w}_k) \in \mathbf{W}_1 \times \mathbf{W}_2 \cdots \times \mathbf{W}_k$ to maximize $\Pr\{(\mathbf{w}_1, \mathbf{w}_2, \cdots, \mathbf{w}_k) \mid (\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_k)\}$, therefore, we can remove the factor $\Pr\{\mathbf{x}_i \mid \mathbf{w}_i\}$ from $\mathrm{P}(\mathbf{w}_i)$ such that

$$\mathrm{P}(\mathbf{w}_i) = \max_{\mathbf{w}_{i-1} \in \mathbf{W}_{i-1}} \Pr\{\mathbf{w}_i | \mathbf{w}_{i-1}\} \, \mathrm{P}(\mathbf{w}_{i-1})$$

and

$$\mathrm{P}(\mathbf{w}_1) = \Pr\{\mathbf{w}_1\}$$

The probabilities $\Pr\{\mathbf{w}_i\}$ and $\Pr\{\mathbf{w}_i \mid \mathbf{w}_{i-1}\}$ are looked up from the dictionaries:

$$\Pr\{\mathbf{w}_i\} = D_w[\mathbf{w}_i]$$

$$\Pr\{\mathbf{w}_i \mid \mathbf{w}_{i-1}\} = D_p[(\mathbf{w}_{i-1}, \sqcup, \mathbf{w}_i)]$$

The derived recurrence suggests that the optimization problem can be mapped to the problem of trellis decoding, which is again solved by dynamic programming. The trellis for our problem has k time stages. The observed codeword at the i-th stage is $\mathbf{x}_i$ for $i \in \{1, \cdots, k\}$. There are $|\mathbf{W}_i|$ vertices at stage i with each representing an element $\mathbf{w}$ of $\mathbf{W}_i$ and being associated with the probability $\Pr\{\mathbf{w}\}$. The weight of the directed edge from a vertex at stage i with word $\mathbf{w}_x$ to a vertex of stage $i+1$ with word $\mathbf{w}_y$ is the conditional probability $\Pr\{\mathbf{w}_y \mid \mathbf{w}_x\}$. An example of the mapping is shown in Figure 4.5. Our target is to compute the sequence which achieves $\max_{\mathbf{w}_k \in \mathbf{W}_k} \mathrm{P}(\mathbf{w}_k)$, which leads to the Viterbi path in the corresponding trellis starting from a vertex in stage 1 and ending at a vertex in stage k.

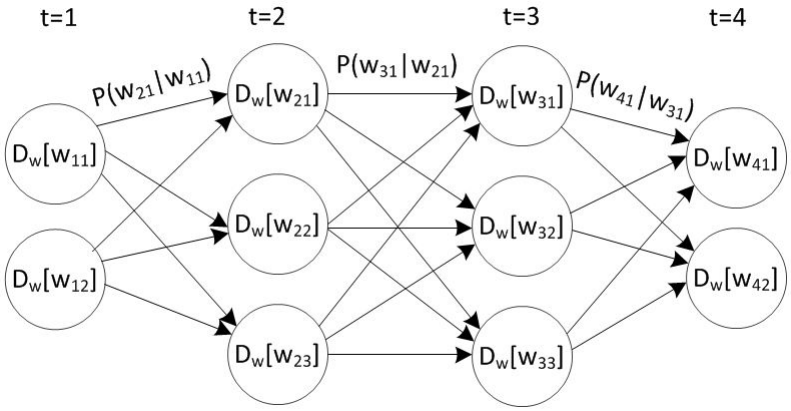


Figure 4.5: An illustrative example of the mapping to trellis decoding. The sets $\mathbf{W}_1 = \{\mathbf{w}_{1,1}, \mathbf{w}_{1,2}\}$, $\mathbf{W}_2 = \{\mathbf{w}_{2,1}, \mathbf{w}_{2,2}, \mathbf{w}_{2,3}\}$, $\mathbf{W}_3 = \{\mathbf{w}_{3,1}, \mathbf{w}_{3,2}, \mathbf{w}_{3,3}\}$ and $\mathbf{W}_4 = \{\mathbf{w}_{4,1}, \mathbf{w}_{4,2}\}$ respectively corresponds to the subcodewords $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$ and $\mathbf{x}_4$.

Algorithm 5 Viterbi$((\mathbf{W}_1, \cdots, \mathbf{W}_k), (\mathbf{x}_1, \cdots, \mathbf{x}_k), D_w, D_p)$

$n \leftarrow \max_{l \in [1,k]} |\mathbf{W}_l|$
Let p and s be two $n \times k$ tables
$p_{\max} \leftarrow 0$, $index \leftarrow 0$
for t from 1 to k **do**
  for i from 1 to $|\mathbf{W}_t|$ **do**
    $p(i,t) \leftarrow D_w[\mathbf{W}_t[i]]$
    $p_{\max} \leftarrow 0$, $index \leftarrow 0$
    for j from 1 to $|\mathbf{W}_{t-1}|$ **do**
      $p' \leftarrow D_p[(\mathbf{W}_{t-1}[j], \sqcup, \mathbf{W}_t[i])] \cdot p[j, t-1]$
      if $p' > p_{\max}$ **then**
        $p_{\max} \leftarrow p'$
        $index \leftarrow j$
    $p(i,t) \leftarrow p_{\max}$
    $s(i,t) \leftarrow index$
$words \leftarrow [\mathbf{W}_k[index]]$
for t from k to 2 **do**
  $i \leftarrow s(index, t)$
  $words$.appendToFront$(\mathbf{W}_{t-1}[i])$
  $index \leftarrow i$
return $words$

The dynamic programming algorithm for solving our trellis decoding problem is specified in Algorithm 5, which is adapted from the Viterbi decoding [56]. The final solution is computed iteratively, starting from $\mathrm{P}(\mathbf{w}_1)$ according to the recurrence. When the last iteration is finished, we trace back along the Viterbi path recorded in the table s, and collect the selected words to form an estimated text $\hat{\mathbf{t}}$. The complexity of the Viterbi decoding algorithm is $\mathcal{O}(n^2k)$, where k is the length of the input codeword list, and $n = \max_{i\in[1,k]} |\mathbf{W}_i|$ is the cardinality of the biggest input word set. The algorithm requires $\mathcal{O}(nk)$ space for storing the tables p and s.

4.3.4 Post Processing

Additional errors may be introduced during codeword segmentation and ambiguity resolution if unknown/rare words or phrases occur in the input codeword. Unknown words (phrases) refer to the words (phrases) that are not in D_w (D_p) and rare words (phrases) mean the words (phrases) that are in D_w (D_p) but with small frequency. Upon meeting an unknown word, the codeword segmentation algorithm tends to split its codeword into subcodewords representing known short words with the space symbol or decode it to be some known words with the same codeword length as the unknown codeword. Such segmentation and ambiguity resolution introduce additional bit errors. We use a simple post-processing step which further reduces the errors by applying the ECC error decoder on the output of our content-assisted decoder (CAD). Since the CAD recovers most of the erasures, the error rate for the correction codeword getting from CAD becomes much smaller than the original channel erasure rate, which is usually under the error capacity of ECC. Moreover, because the noisy codeword only has erasures, the bits with value 0 and 1 are definitely correct. By getting those information, we can modify the ECC error decoder to improve its error correction capacity. In our work, we use an LDPC

code as the error-correcting code and apply iterative belief propagation algorithm to decode. However, we modify the message passing functions for messages from the variable nodes to the check nodes in the following way: if the variable node is 0 (1) in the original noisy codeword, its likelihood to be 0 is always 1 (0) no matter what messages that node receives from check nodes.

4.4 Experiments

In this section, we evaluate the preformance of our proposed content-assisted decoding scheme and discuss the experiment results.

4.4.1 Implementation Detail

Our implementation supports the use of basic punctuations in the input text files, including ',', '.', '?' and '!'. This is done by adding another function in the definition of $\mathrm{c}(i)$. The function measures the number of erasures in the subcodeword that can be decoded as a word followed by a punctuation.

When we estimate the last "space" position for the codeword $\mathbf{x}[i]$, we begin with the last subcodeword of length $k = lmean$, then search subcodeword of length $lmean-1$, $lmean+1$, $lmean-2$, $lmean+2$, $\cdots$, until we find a good last word such that $\mathrm{c}(i) = 0$, where $lmean$ is the mean of length$(\pi_w(\mathbf{w}))$, for all $\mathbf{w} \in D_w$. Because the subcodeword length is near $lmean$ with high probability, this heuristic method can speed up the code segmentation algorithm.

During ambiguity resolution, overflow may occur when the input codeword length is very long due to the multiplications of floating point numbers. We thus use a logarithmic version of the recurrence, which uses additions instead of multiplications of floating point numbers. This significantly delays the overflow.

A smoothing technique is used for computing $\Pr\{\mathbf{w}_i \mid \mathbf{w}_{i-1}\}$. The probability $\Pr\{\mathbf{w}_i\}$ is used if the phrase $(\mathbf{w}_{i-1}, \sqcup, \mathbf{w}_i)$ is unknown to D_p. And if for a word set

$\mathbf{W}_i$, $\forall \mathbf{w} \in \mathbf{W}_i$ is not in the dictionary, we set $\forall \mathbf{w} \in \mathbf{W}_i$, $\Pr\{\mathbf{w}\} = a$, where $0 < a < 1$ is a random number. The reason is that returning 0 for unknown words or phrases suddenly makes the whole joint probability be 0 and cancels the path.

4.4.2 Performance Evaluation

We evaluate decoding performance of our proposed CAD by comparing the bit erasure rates of using LDPC erasure hard decoder alone, the bit error rate of using CAD, and the bit error rate of applying modified LDPC error soft decision decoder on top of CAD. The test inputs include 3 self-collected paragraphs and 24 paragraphs randomly extracted from the Canterbury Corpus, the Calgary Corpus, the Large Corpus [2], and the large text compression benchmark [1] (see Table 4.1). All the testing inputs use basic punctuations. In the future, we would like to support numbers, more punctuations and math symbols. The functions π_s and π_s^{-1} are implemented with Huffman coding. We use a $(3584, 3141)$-random LDPC code.

Table 4.1: The benchmark used in our performance evaluation

Name	**Category**	**From**
`email`	Email discussion	Calgary
`lcet10`	Lecture notes	Canterbury
`alice`	Novel	Canterbury
`conf-intro`	Call for paper	Self-collected
`bible`	The King James version of the bible	Large
`asyoulike`	Shakespeare play	Canterbury
`plrabn`	Poetry	Canterbury
`news`	Web news	Self-collected
`enwiki8`	Wikipedia texts	Large
`world192`	The world fact book	Large

The decoding results for each scheme for $f = 0.1$ is shown in Figure 4.6. The

bit-erasure rate makes the LDPC erasure hard decoder fail to converge with high probability. The results are averaged from 100 experiments. The use of CAD successfully recovers the erasures and brings the number of errors down to make the ECC decoding effective again. The completeness of the dictionaries determines the decoding performances. For instance, in the benchmarks `conf-intro`, `enwiki8`, `plrabn` and `world`, where most of the words or phrases are unknown to the dictionaries, our decoder introduces additional errors by aggressively breaking the codewords of the unknown words with spaces.

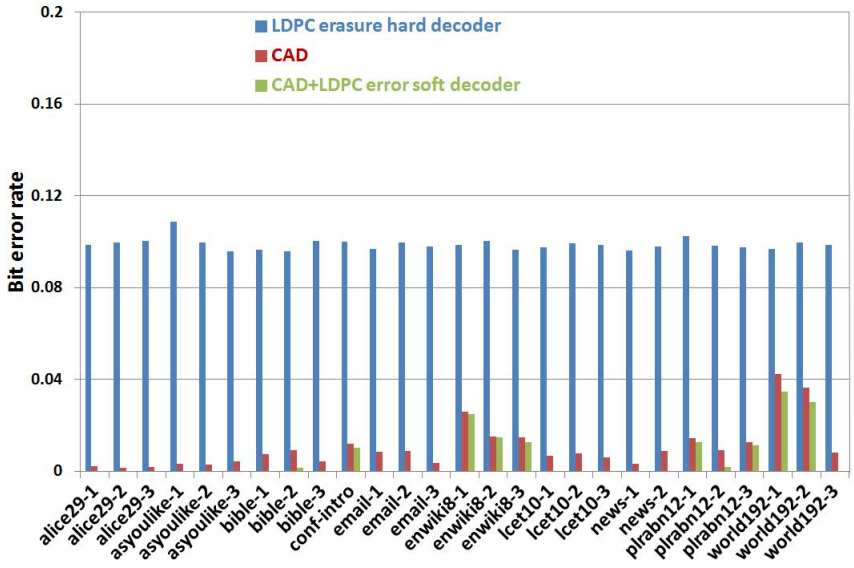


Figure 4.6: The comparison on the correction performance of three decoders: LDPC erasure hard decoder, CAD only and CAD+LDPC error soft decoder.

5. SUMMARIES AND FUTURE DIRECTIONS

This thesis was motivated by the need for effective data representation and coding schemes, which are helpful to efficient and reliable data storage in flash memories based on their unique properties such as: increasing a flash cell level is easy, but decreasing a flash cell level is very costly because of incurring block erasure; flash memories are usually used in low-power embedded devices where energy consumption is the most important factor in the system's performance; flash memories support random and fast access for the data. The topics we have discussed in this work include rank modulation with multiplicity for flash memories, software techniques for reliable embedded flash storage at low voltages and content-assisted file decoding for flash memories. For systems using flash memories, our proposed techniques can extend their longevity and improve their reliability and performance. In this chapter, we summarize our contributions and present suggestions for future work.

5.1 Summaries and Contributions

We have presented a new data representation scheme for flash memoires, which is called rank modulation with multiplicity. It is an extension of rank modulation with the advantages of higher capacity and efficient programming. We have focused on the rewriting of data based on this new scheme and have studied its basic properties, including the rewriting cost, optimal ways to change rank modulation states and the expansion of rank modulation states given the rewriting cost. We have considered both the unweighted and weighted rewriting cost and described the analysis respectively. This scheme can solve both the problem of overshooting while programming cells and the problem of memory endurance in aging devices.

The high voltage requirement of on-chip flash memories is a barrier to reducing the

total energy consumption of low-power devices. We have examined the main factors affecting the behavior of flash memories at low voltage. Based on our observations of flash memory behavior at low voltage, we have proposed three algorithms—in-place writes, multiple-place writes, and RS-Berger codes—that aim to make flash memories available and reliable at low voltage while tolerating the resource limitations of low power devices. Our evaluation shows that in-place writes can save 34% of energy consumption for a sensing workload on the MSP430 microcontroller. Our storage techniques enable battery-powered devices to require fewer or smaller batteries or to become batteryless.

For the sake of reliable file storage in flash memories, we have presented the content-assisted decoder, which makes use of the random and fast access properties of flash memories and the redundancy in the content existing in the text files, to recover the erasures in the codeword. To the best of our knowledge, this is the first decoding scheme for flash memories that is based on looking up the dictionaries for information verification and error/erasure correction. The dictionaries are gained from the statistical properties of words and phrases in the text of a given language. We have designed the dynamic programming algorithms for word segmentation and choosing the most likely word for each segment to form the most likely word sequence as a recovery for the original input text file. We have evaluated the new decoding scheme on a set of benchmark files.

5.2 Future Directions

In order to think further about our research, here, we are interested in discussing potential research work in future.

One of our general objective is to construct rewriting codes for flash memories based on rank modulation with multiplicity proposed in this thesis and explore the

error correcting code for it. It will be interesting to study the following topics about rank modulation with multiplicity in future:

- Analyze the rewriting performance for rank modulation with multiplicity and find some bounds according to the rewriting ball size such as Gilbert type lower bound and sphere packing upper bound.
- Construct good rewriting codes that achieve or near the bounds.
- Define an error model for rank modulation with multiplicity where the number of errors corresponds to the minimal number of adjacent transpositions required to change a given stored permutation to another erroneous one and study its corresponding error-correcting codes.

We have designed the RS-Berger code for reliable flash writes under voltages below the requirement on the specification. Although the RS-Berger code can correct errors dramatically, it consumes much energy due to the very intense computation operations for the Reed Solomon decoding. Future work includes finding more energy-efficient coding schemes to combat flash writes errors caused by low voltage. Currently, the systems cannot take full advantage of dynamic voltage scaling. The new coding schemes should support dynamic voltage adjustment for flash writes and consume less energy.

Another plan is to introduce benchmarks for the storage systems of low power devices. The standard benchmarks that are currently used to evaluate the storage systems are designed for desktop computers and not immediately applicable to the low-power domain.

We have provided content-assisted decoding algorithms for file erasure recovery. The algorithms can be slightly modified and upgraded to support error corrections.

Also, it is very desirable to extend our content-assisted file decoding method to support more general files. In the current stage, decoder only supports plain text files with letters and basic punctuations including ‘,’, ‘.’, ‘?’ and ‘!’. In future, we are planning to support numbers, more punctuations, math symbols and documents with format information. Our final goal for the decoder is to decode more general types of files such as pictures, music and videos. It should find the solutions to the following problems:

- Define the dictionaries and collect data for the dictionaries. Currently, the dictionaries are the statistical properties of words and phrases in the texts. For image, audio or video files, what should their dictionaries include? How to get the dictionaries for those different types of files.

- Construct the source encoder to compress the original files such that we can still make use of the redundancy left in the information bits for content verification. How to compress the format information existing in the documents?

- Design the algorithms to split the image, audio or video files. The text files are segmented by words or phrases. What are the segmentation unit for image, audio or video files?

They all remains as open questions.

REFERENCES

[1] Large Text Compression Benchmark. http://mattmahoney.net/dc/text.html, May 2012.

[2] The Canterbury Corpus Benchmark. http://corpus.canterbury.ac.nz/index.html, May 2012.

[3] D. Agrawal, B. Li, Z. Cao, D. Ganesan, Y.i Diao, and Pr. J. Shenoy. Exploiting the interplay between memory and flash storage in embedded sensor devices. In *RTCSA*, pages 227–236, 2010.

[4] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393 – 422, 2002.

[5] A. Barg and A. Mazumdar. Codes in permutations and error correction for rank modulation. *IEEE Transactions on Information Theory*, 56(7):3158–3165, 2010.

[6] J. M. Berger. A note on error detection codes for asymmetric channels. *Information and Control*, 4(1):68 – 73, 1961.

[7] C. Berrou, A. Glavieux, and P. Thitimajshima. Near shannon limit error-correcting coding and decoding: Turbo-codes. In *Communications, 1993. ICC '93 Geneva. Technical Program, Conference Record, IEEE International Conference on*, volume 2, pages 1064 –1070 vol.2, May 1993.

[8] M. Buettner, B. Greenstein, D. Wetherall, and J. R. Smith. Revisiting smart dust with rfid sensor networks. In *7th ACM Workshop on Hot Topics in Networks*, Oct. 2008.

[9] P. Cappelletti, C.Golla, P. Olivo, and E. Zanoni. *Flash memories*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[10] Y. Cassuto, M. Schwartz, V. Bohossian, and J. Bruck. Codes for asymmetric limited-magnitude errors with application to multilevel flash memories. *Information Theory, IEEE Transactions on*, 56(4):1582 –1595, April 2010.

[11] B. Chen, X. Zhang, and Z. Wang. Error correction for multi-level nand flash memory using reed-solomon codes. In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, pages 94 –99, Oct. 2008.

[12] S. Chen. What types of ECC should be used on flash memory? Application note for SPANSION. http://www.spansion.com/Support, March 2011.

[13] M. Fujino and V. G. Moshnyaga. An efficient hamming distance comparator for low-power applications. In *Electronics, Circuits and Systems, 2002. 9th International Conference on*, volume 2, pages 641 – 644, 2002.

[14] E. En Gad, A. Jiang, and J. Bruck. Trade-offs between instantaneous and total capacity in multi-cell flash memories. In *ISIT*, pages 990–994, 2012.

[15] E. En Gad, M. Langberg, M. Schwartz, and J. Bruck. Generalized gray codes for local rank modulation. In *ISIT*, pages 874–878, 2011.

[16] E. En Gad, M.l Langberg, M. Schwartz, and J. Bruck. Constant-weight gray codes for local rank modulation. *IEEE Transactions on Information Theory*, 57(11):7431–7442, 2011.

[17] E.l En Gad, A. Jiang, and J. Bruck. Compressed encoding for rank modulation. In *ISIT*, pages 884–888, 2011.

[18] E. Gal and S.Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2):138–163, June 2005.

[19] R. Gallager. Low-density parity-check codes. *Information Theory, IRE Transactions on*, 8(1):21 –28, Jan. 1962.

[20] B. Godard, J. M. Daga, L. Torres, and G. Sassatelli. Hierarchical code correction and reliability management in embedded nor flash memories. In *Test Symposium, 2008 13th European*, pages 84 –90, May 2008.

[21] A. L. Goldberger, L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. C. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C. K. Peng, and H. E. Stanley. Physiobank, physiotoolkit, and physionet: Components of a new research resource for complex physiologic signals. *Circulation*, 101(23):e215–e220, 2000.

[22] S. Gregori, A. Cabrini, O. Khouri, and G. Torelli. On-chip error correcting techniques for new-generation flash memories. *Proceedings of the IEEE*, 91(4):602 – 616, April 2003.

[23] L. M. Grupp, J. D. Davis, and S. Swanson. The bleak future of nand flash memory. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[24] Texas Instruments Incorporated. MSP430 ultra-low power microcontrollers. http://www.ti.com/msp430, May 2010.

[25] A. Jiang. On the generalization of error-correcting wom codes. In *ISIT*, pages 1391 –1395, June 2007.

[26] A. Jiang and J. Bruck. *Data Storage*. In-Tech Publisher, New York, USA, 2010.

[27] A. Jiang, R. Mateescu, M. Schwartz, and J. Bruck. Rank modulation for flash memories. In *ISIT*, pages 1731–1735, 2008.

[28] A. Jiang, M. Schwartz, and J. Bruck. Error-correcting codes for rank modulation. In *ISIT*, pages 1736–1740, 2008.

[29] A. Jiang, M. Schwartz, and J. Bruck. Correcting charge-constrained errors in the rank-modulation scheme. *IEEE Transactions on Information Theory*, 56(5):2112–2120, 2010.

[30] A. Jiang and Y. Wang. Rank modulation with multiplicity. In *Proceedings of IEEE Workshop on Application of Communication Theory to Emerging Memory Technologies (ACTEMT)*, pages 1928–1932, 2010.

[31] T. Kløve. Lower bounds on the size of spheres of permutations under the chebychev distance. *Des. Codes Cryptography*, 59(1-3):183–191, 2011.

[32] T. Kløve, T. Lin, S. Tsai, and W. Tzeng. Permutation arrays under the chebyshev distance. *IEEE Transactions on Information Theory*, 56(6):2611–2617, 2010.

[33] C. Lee, T. Lin, M. Shieh, S. Tsai, and H. Wu. Decoding permutation arrays with ternary vectors. *Des. Codes Cryptography*, 61(1):1–9, 2011.

[34] Y. Li, Y. Wang, A. Jiang, and J. Bruck. Content-assisted file decoding for nonvolatile memories. In *Proceedings of the 46th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, USA, 2012.

[35] S. Lin and D. J. Costello Jr. *Error Control Coding - Fundamentals and Applications*. Prentice Hall computer applications in electrical engineering series. Prentice Hall, Upper Saddle River, NJ, USA, 1983.

[36] D. J. C. MacKay and R. M. Neal. Near shannon limit performance of low density parity check codes. *Electronics Letters*, 32(18):1645, Aug. 1996.

[37] A. M. Mainwaring, J. Polastre, and R. Szewczyk. Wireless Sensor Networks for Habitat Monitoring. In *Mobile Computing and Networking*, pages 88–97, 2002.

[38] D. Malan, T. Fulford-Jones, M. Welsh, and S. Moulton. CodeBlue: An Ad Hoc Sensor Network Infrastructure for Emergency Medical Care. In *Wearable and Implantable Body Sensor Networks*, 2004.

[39] G. Mathur, P. Desnoyers, D. Ganesan, and P. J. Shenoy. Ultra-low power data storage for sensor networks. In *IPSN*, pages 374–381, 2006.

[40] Microchip. 32-bit PIC MCUs. http://www.microchip.com/en_US/family/pic32, June 2010.

[41] V. Papirla and C. Chakrabarti. Energy-aware error control coding for flash memories. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 658–663, New York, NY, USA, 2009.

[42] P. Pavan, R. Bez, P. Olivo, and E. Zanoni. Flash memory cells - an overview. *Proceedings of the IEEE*, 85(8):1248–1271, 1997.

[43] J. Polastre, R. Szewczyk, and D. Culler. Telos: enabling ultra-low power wireless research. In *Information Processing in Sensor Networks, Fourth International Symposium on*, pages 364 – 369, April 2005.

[44] I. S. Reed and G. Soloman. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[45] R. L. Rivest and A. Shamir. How to Reuse a "Write-Once" Memory. *Information and Computation/information and Control*, 55:1–19, 1982.

[46] M. Salajegheh, S. Clark, B. Ransford, K. Fu, and A. Juels. Cccp: secure remote storage for computational rfids. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 215–230, Berkeley, CA, USA, 2009. USENIX Association.

[47] M. Salajegheh, Y. Wang, K. Fu, A. Jiang, and E. Learned-Miller. Exploiting half-wits: smarter storage for low-power devices. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pages 4–4, Berkeley, CA, USA, 2011. USENIX Association.

[48] A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R. Smith. Design of an rfid-based battery-free programmable sensing platform. *Instrumentation and Measurement, IEEE Transactions on*, 57(11):2608 –2615, Nov. 2008.

[49] M. Schwartz. Constant-weight gray codes for local rank modulation. In *ISIT*, pages 869–873, 2010.

[50] M. Schwartz and I. Tamo. Optimal permutation anticodes with the infinity norm via permanents of (0, 1)-matrices. *J. Comb. Theory, Ser. A*, 118(6):1761–1774, 2011.

[51] M. Shieh and S. Tsai. Decoding frequency permutation arrays under chebyshev distance. *IEEE Transactions on Information Theory*, 56(11):5730–5737, 2010.

[52] V. Shnayder, B. R. Chen, K. Lorincz, T. R. F. F. Jones, and M. Welsh. Sensor networks for medical care. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, SenSys '05, pages 314–314, New York, NY, USA, 2005.

[53] Atmel AVR Solutions. ATmega128L. http://www.atmel.com/atmel/acrobat/doc2467.pdf, July 2010.

[54] I. Tamo and M. Schwartz. Correcting limited-magnitude errors in the rank-modulation scheme. *IEEE Transactions on Information Theory*, 56(6):2551–2560, 2010.

[55] S. Tsai and M. Shieh. Decoding frequency permutation arrays under infinite norm. In *ISIT*, pages 2713–2717, 2009.

[56] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *Information Theory, IEEE Transactions on*, 13(2):260 – 269, April 1967.

[57] Z. Wang, A. Jiang, and J. Bruck. On the capacity of bounded rank modulation for flash memories. In *ISIT*, pages 1234–1238, 2009.

[58] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. Microhash: An efficient index structure for flash-based sensor devices. In *FAST*, pages 31–44, 2005.

[59] G. Zemor and G. D. Cohen. Error-correcting wom-codes. *Information Theory, IEEE Transactions on*, 37(3):730 –734, May 1991.

[60] H. Zhou, A. Jiang, and J. Bruck. Error-correcting schemes with dynamic thresholds in nonvolatile memories. In *ISIT*, pages 2109–2113, Aug. 2011.

[61] H. Zhou, A. Jiang, and J. Bruck. Nonuniform codes for correcting asymmetric errors. In *ISIT*, pages 1011–1015, Aug. 2011.

[62] H. Zhou, A. Jiang, and J. Bruck. Systematic error-correcting codes for rank modulation. In *ISIT*, pages 2978–2982, 2012.